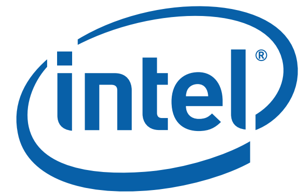
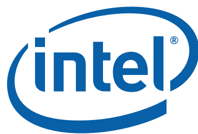


Intel® Infrastructure DSP Solution Version 1.2

Programmer's Guide

November 2007



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Intel Corporation may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights that relate to the presented subject matter. The furnishing of documents and other materials and information does not provide any license, express or implied, by estoppel or otherwise, to any such patents, trademarks, copyrights, or other intellectual property rights.

MPEG is an international standard for video compression/decompression promoted by ISO. Implementations of MPEG CODECs, or MPEG enabled platforms may require licenses from various entities, including Intel Corporation.

This Programmer's Guide as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Intel® Infrastructure DSP Solution may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

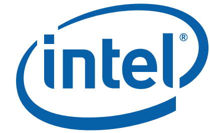
Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at http://www.intel.com.

Intel, Intel logo, and Intel XScale are trademarks of Intel Corporation in the U.S. and other countries.

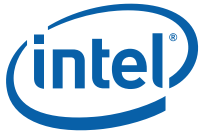
*Other names and brands may be claimed as the property of others.

Copyright © 2007, Intel Corporation. All Rights Reserved.



Contents

1.0 Introduction 7
1.1 Scope 7
1.2 Audience 7
1.3 Related Documents 7
1.4 Terminology 7
1.5 General 8
2.0 Architectural Overview 10
3.0 Intel® Infrastructure DSP Solution Run-Time Interfaces 14
3.1 Control Interface 14
3.2 PCM Data Interface 15
3.3 Packet Interface 17
4.0 Components, Features, and Parameters 18
4.1 Network Endpoint 18
4.2 Encoder 21
4.3 Decoder 26
4.4 Tone Generator 31
4.5 Tone Detector 32
4.6 Audio Player 33
4.7 Audio Mixer 34
4.8 Audio Stream Router 35
4.9 T.38 Fax 37
4.10 Message Agent 38
5.0 Programming Guide 39
5.1 Initialization 39
5.2 Programming Model 40
6.0 OS-Specific Issues 42
6.1 Linux* 42
7.0 User-Defined Messages 44
7.1 Overview 45
7.2 Pre-Defined User Messages 47
7.2.1 Link Message 49
7.2.2 Link Break Message 50
7.2.3 Link Switch Message 50
7.2.4 Start IP Message 51
7.2.5 Stop IP Message 52
7.2.6 Set Up Call Message 52
7.2.7 Set Call Parameters Message 53
7.2.8 Set Up Call with Parameters Message 54
7.2.9 Switch Call Message 55
7.2.10 Create Three-Way Call Message 56
7.2.11 Exit Three-Way Call Message 57
7.2.12 Teardown Three-Way Call Message 57
7.2.13 Back to Two-Way Call Message 58
7.2.14 Set Clear Channel Message 59
7.2.15 T.38 Switchover Message 60
7.2.16 Set Parameters Message 60
7.3 Pre-Defined User-Response Messages 61
7.3.1 Acknowledge Message 61



7.3.2 Stop Acknowledge Message 62

8.0 Application Examples 62

8.1 IP Interface....62

8.2 Caller-ID Generator 64

Figures

1 Architecture of the Intel® Infrastructure DSP Solution 11

2 Data-Flow and Data-Processing Functions 12

3 Intel® Infrastructure DSP Solution Message, Data, and Tasks....13

4 Control Interface and Message Queues 15

5 PCM Data Interface....17

6 Packet Interface 18

7 Example of Tone Disabler Event and Event Handling in Fax over IP application 20

8 G.729.1 RTP Payload Header....23

9 G.729.1 Bitrate Negotiation through RTP Payload....25

10 Jitter Buffer Statistics Resetting Mechanism 29

11 Audio Stream Connections in a Three-Way Call....35

12 Terminations and Router 36

13 General State-Machine Approach for Client Applications....42

14 Intel® Infrastructure DSP Solution Client Driver in Linux* 44

15 Decoding User-Defined Messages in the Message Agent....47

16 Intel® Infrastructure DSP Solution Application in Linux* 64

17 Snap-shot of Demo Codelet Make File 69

18 Snap-shot of PlugInConfig.c file Showing Changes Needed for Unplugging a Third Party Plug-in....71

19 Example to Remove a Plug-in in Codelet Demo Make File....72

20 Socket Interface....73

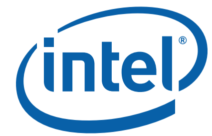
Tables

1 MBS Table 24

2 FT Table....24

3 Linux* User Mode DSP Library Threads....43

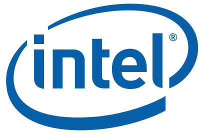
4 Linux* User Mode DSP Application Threads 43

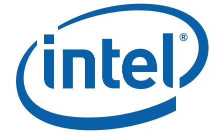


Revision History

Date	Revision	Description
November 2007	003	Updated with G.729.1-related information
August 2007	002	Updated for the Intel® IXP43X product line of network processors
July 2007	001	Initial release

§ §





1.0 Introduction

The Intel® Infrastructure DSP Solution Version 1.2 is a set of software modules that provides basic voice and signal processing functionality for Voice-over-Internet-protocol (VoIP) on the Intel® IXP43X product line of network processors.

This document explains the Intel® Infrastructure DSP Solution architecture and provides guidelines and examples to application developers.

1.1 Scope

The *Intel® Infrastructure DSP Solution Version 1.2 Programmer's Guide* specifies how you can interface to the DSP solution. This document provides more application information on how the interface can be effectively used. Some examples are given for illustration purposes. Details on pre-defined user messages, which are not part of the core DSP solution, are provided to help ease integration.

1.2 Audience

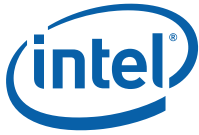
This document is intended for third-party software developers who are using the DSP solution to build Integrated Access Devices (IADs) such as Customer Premises Equipment (CPE). It is assumed that the reader has general knowledge of VoIP applications and produts.

1.3 Related Documents

Title
Intel® Infrastructure DSP Solution Version 1.2 API Reference Manual
Intel® Infrastructure DSP Solution Version 1.2 Codelet Demo Guide
Intel® Infrastructure DSP Solution Version 1.2 Release Notes

1.4 Terminology

Term	Description
AEC	Acoustic Echo Canceller (Note: Not Supported)
AGC	Automatic Gain Control for voice data towards IP network
ALC	Automatic Level Control
CPE	Consumer Premise Equipment
CNG	Comfort Noise Generator
DEC	Decoder
EC	Echo Cancellation
FSK	Frequency Shift Keying
FT	Frame Type
G3	Group 3
HPF	High Pass Filter
HSS	High Speed Serial



Term	Description
IAD	Integrated Access Device
ID	Identification
iLBC	internet Low Bitrate Codec
IP	Internet Protocol
ISR	Interrupt Service Routine
LEC	Line Echo Canceller
MBS	Maximum Bitrate Supported
NPE	Network Processing Engine
NLP	Non-linear Processing (for EC)
OSAL	Operating System Abstraction Layer
PCM	Pulse Code Modulation
PLC	Packet Loss Concealment
RTP	Real-time Transport Protocol
SLIC	Subscriber Line Interface Circuit
SNR	Signal to Noise Ratio
SP	Signal Processing
SRTP	Secure Real-time Transport Protocol
TD	Tone Detector
TDM	Time Division Multiplexing
TG	Tone Generator
USCI	Unified Speech CODEC Interface
VAD	Voice Activity Detection

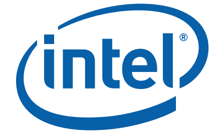
1.5 General

The Intel® Infrastructure DSP Solution is designed for audio processing, and is targeted for next-generation IADs such as CPE. Specifically, the software is tailored to perform audio compression, echo cancellation, tone processing, and jitter control required in IP media gateway or real-time media-streaming functionalities.

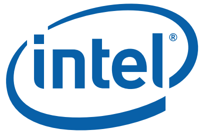
This document describes the control and data interfaces in order for a third-party developer to incorporate the DSP solution into a media gateway and integrate it with other client software. Together with the *Intel® Infrastructure DSP Solution Version 1.2 API Reference Manual*, this document should provide sufficient details of the interfaces and message and data-delivery mechanisms so that user applications can fully configure and control processing operations and services.

This release of the DSP solution supports the following features:

- Line echo cancellation up to 128ms for narrowband or up to 64 ms for Wideband
- G.711 µ-law and A-law CODEC with VAD and CNG support
- G.723.1 and G.729ab with VAD and CNG support
- G.726 with 16, 24, 32 and 40 Kbps rates and RFC3551 and I.366 Annex E packing formats
- G.722 Wideband codec
- G.729.1 Wideband/Narrowband CODEC with bitrate negotiation (Bitrates Supported : 8K. 12k, 14K, 16K, 18K, 20K, 22K, 24k, 26K, 28k, 30K, and 32Kbps)



- internet Low Bitrate CODEC (iLBC) with dual frame size support : 20ms and 30 ms
- T.38 Fax relay with V.17, V.29, V.27Ter, and V.21 fax modulation/ demodulation support
- Dynamic switching of codec on the fly with automatic switching of decoder types according to the received RTP packets
- Packet loss concealment (PLC) for G.711, G.726, and G.722
- Configurable PCM interface in the Wideband or Narrowband mode
- Support multiple frames per packet. Maximum numbers of frames per packet are:
    - 6 for G.711 and G.722
    - 8 for G.723
    - 9 for G.726 40 Kbps
    - 12 for G.726 32 Kbps
    - 12 for G.726 24 Kbps
    - 12 for G.729ab and G.726 16 Kbps
    - for G.729.1 maximum MFPP varies with bitrate (8K: 24, 12K: 17, 14: 14, 16K: 12, 18K: 11, 20K: 10, 22K: 9, 24K: 8, 26K: 7, 28K: 7, 30K: 6, 32K: 6)
    - 10 for iLBC-30ms and 13 for iLBC-20ms
- Dynamically changing the frames per packet on the fly
- Automatic Gain Control (AGC) support for encoder, with provision for manual setting with mute
- Automatic Level Control (ALC) support for decoder, with provision for manual setting with mute
- DTMF generation and detection
- Modulated-tone generation capability
- Tone Disabler in NET component. Detects 2100 Hz tone with periodic phase-reversals and report events on Tx and Rx direction separately
- T30 Preamble Detector in NET component. It reports T30 preamble events on both Tx and Rx direction separately
- APIs in NET component for ToneDisabler for user programmable silence threshold level and silence duration
- APIs to re-enable reporting of events "phase-reversal in 2100 Hz tone" and "T30 Preamble" for both Tx & Rx direction
- Detection and generation of user-specified tones
- FSK modem signal generating and receiving for caller ID
- US, China, and Japan call-progress tone generation
- Dynamic DTMF tone clamping
- RFC 2833 tone event support for DTMF with variable frame rate
- Dynamic/Adaptive Jitter Buffer algorithm
- Additional statistics required for extended report as per RFC3611. Statistics provided are Maximum jitter, Minimum jitter, Mean jitter, Standard deviation, Jitter buffer Maximum delay, Jitter buffer absolute maximum delay, Jitter buffer nominal delay, Jitter discard rate and Echo Return Loss Enhancement (ERLE)
- Audio mixer component is enhanced to support up to four Conference Calls simultaneously depending on configuration



- Audio player for voice prompts, on-hold music, and so on. (playing back G.711 or G.729ab encoded data)
- Low-latency TDM switching
- Digital gain control at the front end
- User-defined control interface
- Lip-sync delay control
- Plug-in interface for pluggable modules

Note: AEC component in Network Endpoint MPR is not supported.

2.0 Architectural Overview

The DSP solution is implemented as an independent module having its own tasks and runtime environment. The software architecture is of a two-layer hierarchy - a control layer that handles the control interface and control logic and a data-processing layer where the media data streams are processed by appropriate algorithms.

Figure 1 shows the logic decomposition of the DSP solution modules.

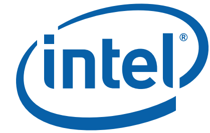


Figure 1. Architecture of the Intel® Infrastructure DSP Solution

From the control point of view, DSP solution channel consists of a set of Media Processing Resource (MPR) components. Each MPR is an addressable entity and can be controlled independently. That gives the maximum flexibility of setting up a channel with various resource configurations, for example, half-duplex call or asymmetry Rx and Tx codec types (Tx codec type and Rx codec type is different for a single channel), if necessary. Software developer has the flexibility to use Intel provided Media processing algorithmic modules or plug-in external algorithmic modules.

These modules are static libraries and can be plugged into the framework during the build process. The Plug-in interface is the interface between the framework and the plug-in modules. The plug-in interface supports the Unified Speech Codec Interface (USCI). For more details on adding Intel provided algorithm (which is in USCI), refer to Appendix A.

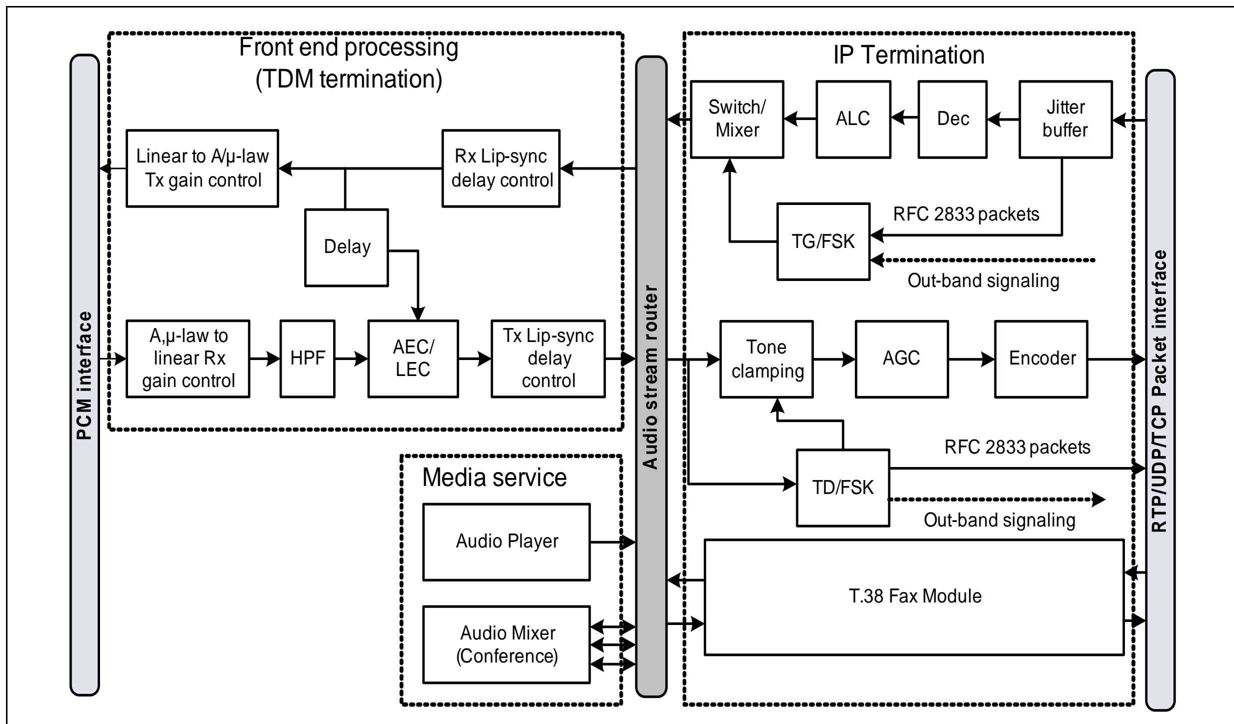
From the perspective of data flows, the data processing functions are depicted in Figure 2. All the functions are executed by real-time tasks (or threads) created during initialization. There is one task for each unique coder frame rate. Currently there is a

10-ms task for G.711, G722, G726 and G.729ab coders, a 20-ms task for G.729.1 and a 30-ms task for the G.723 coder and T.38 Fax. iLBC runs on both 20-ms and 30-ms tasks to support dual frame size.

The 10-ms task also handles all other non-coder voice processing, such as echo cancellation and tone detection. The real-time tasks are of higher priority than the control task and are synchronized (triggered) by the Network Processing Engine (NPE) of the High Speed Serial (HSS) port in the Intel® IXP4XX product line of network processors.

Some of the necessary input and output functions are also performed in the context of the real-time tasks. This includes buffer reading/writing PCM data to and from the PCM interface (the HSS interface is the PCM interface shown in Figure 2), and the external function registered to DSP solution to encode the DSP solution's packets into RTP format for forwarding to the IP stack.

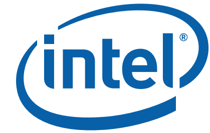
Figure 2. Data-Flow and Data-Processing Functions



The relation among the messages, data and tasks inside and outside the DSP solution, is illustrated by Figure 3, and can be summarized as:

- The control task is driven by the user application (IXP4XX Client Application).
- The real-time tasks are synchronized with the data from the PCM Data Interface. Unblocking of the DSP solution library's read function triggers the real-time tasks according to the algorithms executed by the tasks.
- Real-time tasks generate and consume the encoded audio packets at the fixed rates essentially synchronized with PCM data.
- The encoded audio packets arrive at variable rate asynchronously with the real time tasks.

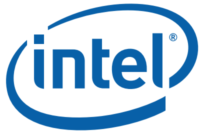
Note: It is important to understand that the internal, real-time tasks are characterized by their hard task deadlines. That means if a real task cannot finish its processing before



the next task period, data will be lost and consequently voice quality is degraded seriously. That may happen if the real-time task is preempted by ISR or other tasks for a long time or simply the processor is overloaded.

Figure 3. Intel® Infrastructure DSP Solution Message, Data, and Tasks

- Application Interface provides interfaces for applications to configure and control the behavior of DSP solution.
- OSAL Interface provides abstractions to all OS specific function calls. This interface also abstracts the kernel space code or user space code based on preprocessor definition.
- Packet Interface provides interface between DSP library and application to transmit and receive voice packets.
- Plug-in Interface enables media processing components to plugged into DSP framework, it provides following capabilities:
  - — Get component identification, Memory requirements, type, version, capabilities and others
  - — Initialization and re-initialization routines
  - — Execute
  - — Get/Set routines for the attributes of components at run time.



The DSP solution uses USCI as the plug-in interface. USCI (Unified Speech Codec Interface) is the interface between the core framework and plug in Codecs/Algorithms. The purpose of this interface is to provide unified access to a module independent of algorithm internals and to enable binaries to be easily integrated into existing software applications. De-coupling the USC interface from the algorithm details provides a high level of independence between the development of system components and the algorithm implementation. Refer to Appendix A and the *Intel® Infrastructure DSP Solution Version 1.2 Release Notes* for information on the usage of USCI.

- PCM Interface defines the generic interface for both Transmit and Receive side of PCM data. This supports the following formats:
  - — Narrowband (8kHz) G.711 A-law 8bit
  - — Narrowband (8kHz) G.711 µ -law 8bit
  - — Wideband (16kHz) 16bit Linear PCM

3.0 Intel® Infrastructure DSP Solution Run-Time Interfaces

The DSP solution is implemented as an independent module executed by its own tasks. User applications do not directly access the internal functions or data.

The DSP solution provides three interfaces for the applications to communicate control information, PCM data, and encapsulated voice packets, respectively, in run-time as shown in Figure 1 and Figure 2.

3.1 Control Interface

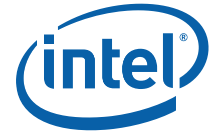
The applications primarily communicate with the DSP solution through the control interface defined as a set of functions, messages and macros.

There are two message queues in the control interface for the in-bound messages from applications to the DSP solution and the out-bound messages in the other direction. Refer to Figure 4. The DSP solution has set the message queue's (in-bound/out-bound) length to 64. Two interface functions, `xMsgSend()` and `xMsgReceive()`, can be used for the application to send and receive messages to/from the queues, respectively.

The DSP solution spawns a dedicated control task pending on the in-bound message queue to handle the control messages. The reason for isolating the DSP solution from user applications by message queues is to avoid the internal control functions being accessed by multiple tasks of the user application, since making the control functions multitask-safe creates extra complexity and subsequent performance penalties.

The DSP solution sends replies or events to the application through the out-bound message queue. The application can retrieve the messages using `xMsgReceive()`. The caller's task of `xMsgReceive()` will be blocked forever (or until timeout) if the out-bound queue is empty.

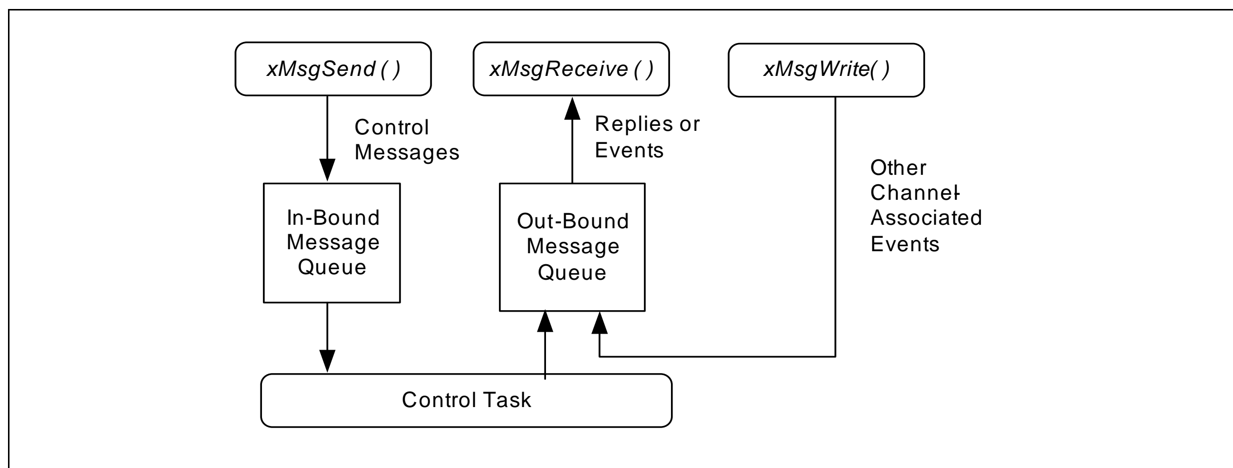
A third function for the control interface, `xMsgWrite()`, allows the application to directly post external messages to the out-bound message queue back to the user application if necessary. This enables the user application to receive all channel-associated events from one place, even though some of these events are external to the DSP solution. For instance, the application may hook a callback function to the ISR that reports the SLIC interface on/off hook events. In the callback function, an external event message as defined by the user is sent to the out-bound message queue to signal the event to the user application.



Both in-bound queue and out-bound queue have queue length of 64. Because of the limitation of the queue lengths, the queues may overflow and the messages may be lost if the application keeps sending messages without waiting for the replies. In this case, the in-bound queue may overflow if the user application is of higher priority than the DSP solution control task, or the out-bound queue may overflow if the user application has lower priority.

Copy-based message delivery is used. That is, the entire message context is actually copied from the deliverer to the receptor rather than passing a pointer around. This avoids dynamically allocating memory for the messages. Since no memory is shared between the DSP solution and the application, the application can reuse the memory of a message for any other purpose immediately after the message is sent. On the other hand, to receive a message the application is responsible for preparing the memory that must be able to accommodate the maximum message size with the alignment at 4-byte boundary.

Figure 4. Control Interface and Message Queues



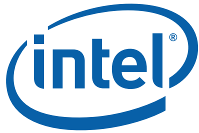
The message format consists of an 8-byte message header plus an optional message payload. The message header contains the common information like channel ID, MPR ID, type, size, etc. A 4-byte transaction ID is provided to allow the user application to keep track of the replies or events. When the DSP solution sends a reply or event message to the user application, it copies the transaction ID from the associated message originated from the user application. Refer to the *Intel® Infrastructure DSP Solution Version 1.2 API Reference Manual* for details of the control message format.

3.2 PCM Data Interface

PCM data represents the audio data stream between the DSP solution and the telephone interface. The easiest way to do this is to use the TDM bus of the Intel® IXP4XX product line of network processors, which is also called HSS device.

The DSP solution is provided with read and write functions to read and write from HSS device. In Linux*, this is done using HSS device driver. Read function call is a blocking call, it is blocked until the number of bytes mentioned in the read call is available in HSS buffer. The user application, however, controls how the HSS is being configured, by parameters being passed to the HSS driver during initialization. HSS driver is the interface between the DSP solution and the HSS NPE for IXP4XX product line processors. HSS driver provides the following APIs to the DSP application:

- `IxHssDriverHssPortInit(hss_port_config)` - This API downloads NPE microcode image corresponding to `npe_image_id` provided in `hss_port_config`



structure and initializes NPE A. Initializes the HSS device. It configures the HSS port for the channelized service and connects the configured port to the HSS driver.

DSP solution configures HSS device using the above APIs.

Figure 5 illustrates the PCM data interface and the data flow between telephone interface and the DSP solution. From the user application's perspective, the HSS can be viewed as a piece of hardware to be properly configured, to interoperate with the external, customer-specific interface connected to it. Once it is configured and started, there is no further user application involvement.

The user application configures the HSS by specifying the signal format to be presented on TDM bus of the HSS device, including the clock rate, time slots, frame sync, endian, and so on. This information is provided through the `HSS_config` structure. For more information on this structure refer to the *Intel® Infrastructure DSP Solution Version 1.2 API Reference Manual*.

Using this set of information, the DSP solution initializes the HSS interface and starts data transfers.

The DSP solution supports dual-band PCM interface over the HSS. In the narrowband mode, the PCM data format is 8-bit A-law or µ-law compressed data at an 8 KHz sampling rate. In the wideband mode, it is 16-bit linear data at 16 KHz sampling rate. To share the TDM bus of the HSS, a wideband audio channel takes four time slots at the 8 KHz frame rate. In the narrowband mode, an audio channel takes one time slot of the TDM bus of the HSS.

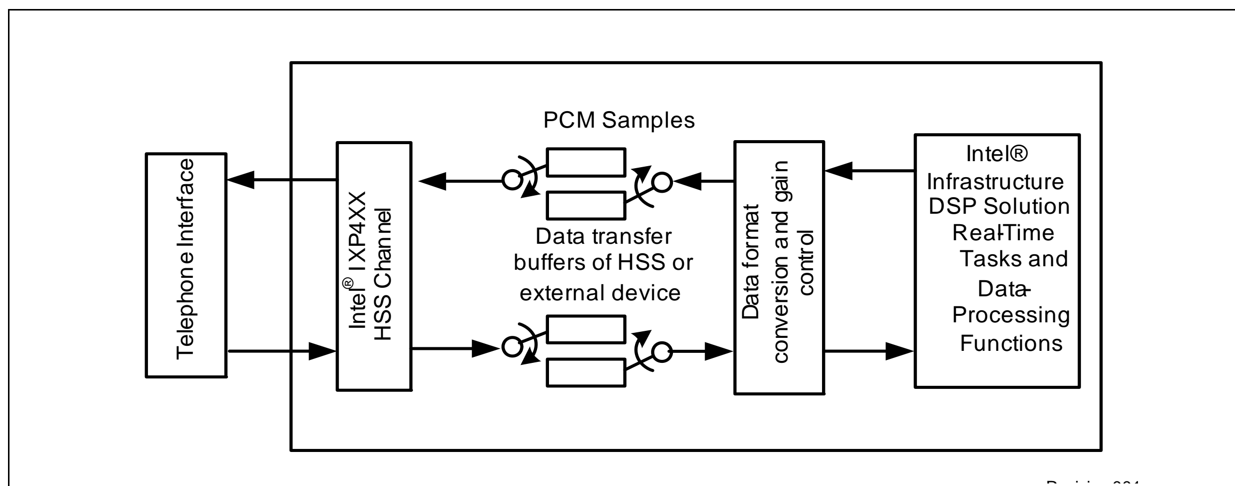
The user applications must specify how those time slots are located if a channel is configured to wideband mode during the system initialization. Sampling rate conversion (SRC) is applied automatically if a wideband channel is connected to a narrowband media processing resource or vice versa. The superior voice quality can be expected only when both the interface and the resources operate in the wideband mode.

The user application may enable more HSS time slots than the number of channels supported by the DSP solution. Four channels are supported by the DSP solution. In this case the time slots are connected to the channels from the first one sequentially and the extra time slots are ignored by default, or you can specify which time slots will be used. To use the low latency time slot switch feature, at least eight time slots must be enabled. The number of time slots can be mentioned in the `numChannelised` member of the `hssportTx_Rx_config` of the `Hss_config` structure.

For more information on this structure refer to the *Intel® Infrastructure DSP Solution Version 1.2 API Reference Manual*.

Internally, the real-time tasks are synchronized with HSS data transfer - the scheduler being signaled by unblocking the read call by the HSS driver each time when certain amount (10 ms) of data is transferred. The real-time tasks may not be invoked at all if the HSS interface is not configured and started properly.

Figure 5. PCM Data Interface



3.3 Packet Interface

Compared to PCM Data Interface, the Packet Interface is a pure software protocol that defines how the encoded audio data packets are exchanged between the DSP solution and the IP interface.

There are two functions and a packet format involved in the Audio Packet Interface as shown in Figure 6. The DSP solution defines the packet format and provides the packet receive function. The user application is responsible for providing the transmit function.

In ingress (packets coming from the IP interface), the IP interface converts each incoming VoIP packet it receives to a DSP solution data packet and then calls `xPacketReceive()` to deliver it to the DSP solution. The user application needs to decode the incoming IP packets to forward the RTP packet payloads with the proper DSP solution header format, with the extracted RTP timestamp, to the proper DSP solution channel.

The function `xPacketReceive()` copies the packet to the jitter buffer without further processing. Therefore `xPacketReceive()` can be called from an Interrupt Service Routine context but re-entry is not allowed. Since the packets are copied by the DSP solution, the caller of the `xPacketReceive()` can free or reuse any memory it may have allocated to buffer the incoming RTP packets upon return from the function.

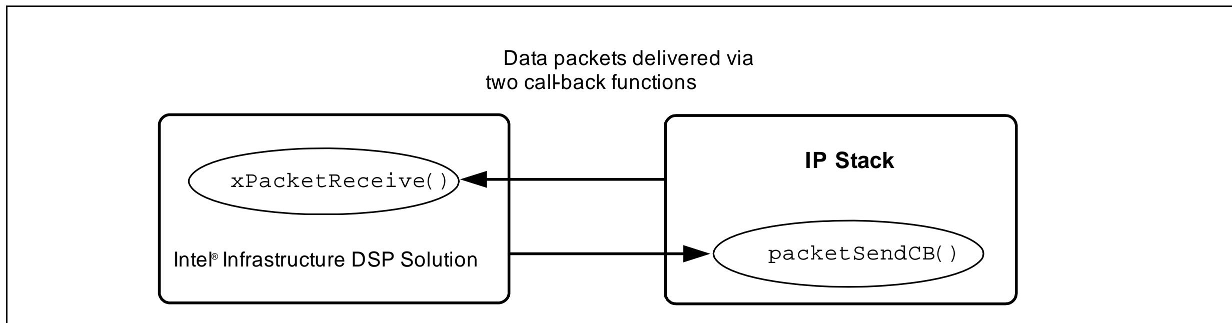
In egress (packets going to the IP interface), through `xDspSysInit()`, the application registers a callback function with the DSP solution. This callback function is supposed to deliver the data packet to the IP interface and sends it out. The DSP solution always prepares the memory for the packet and fills the packet header information (including local time stamp) and packet payload before it calls the function. This user-provided function should create and encode the RTP header with the time stamp in the data packet supplied by the DSP solution. After returning from the function, the DSP solution will immediately reuse the memory for other purpose. Therefore, it may be necessary for the callback function to make a copy of the packet.

Since the function is called from the internal real-time tasks at regular basis each time when a packet is generated, there are two additional requirements for the callback function:

- It must finish as soon as possible without any blocks inside (to allow real-time data to be acquired and processed without data loss)

- It must be multitask-safe (it must allow re-entry)

Figure 6. Packet Interface



4.0 Components, Features, and Parameters

A DSP solution channel consists of several media processing resource (MPR) components, which can be addressed independently by the application. Each component has its particular processing functions and features that are controlled by the messages and parameters. In this section, we will discuss the MPR components and their features and parameters.

Note: AEC component in Network Endpoint is not supported.

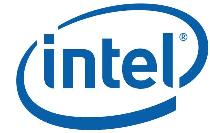
4.1 Network Endpoint

Network Endpoint component is a front-end data processing unit connecting the HSS interface to the rest of MPR components. In addition to receiving and transmitting data, it also applies the gain, A-law or µ-law conversion (in the narrowband mode) in both directions and high-pass filter (HPF) and echo cancellation in the Rx direction (from the HSS to the DSP solution).

The channels of Network Endpoint can be configured to narrowband or wideband during initialization.

In the narrowband mode, the application can specify A-law or µ-law conversion by setting the parameter `XPARMID_NET_LAW`. If this parameter is set to `XPARM_NET_PASSTHRU`, all the front-end processing mentioned above will be automatically bypassed. This is only used for debugging purposes and should not be set in normal applications. When `XPARM_NET_PASSTHRU` is set, the encoder and decoder should also be set to `PASSTHRU` CODEC. In this mode, 8-bit to 16-bit data conversion from HSS to linear is also bypassed and MPR components - such as tone detection and tone generation - are no longer meaningful. This parameter only applies to the narrowband mode.

Digital gain control can be applied to the audio signal in front of the Network Endpoint via the `XPARMID_NET_GAIN_RX` and `XPARMID_NET_GAIN_TX` parameters. Because it takes extra processing time and may also affect voice quality if not set properly, this feature should be used only if the gain control is not available in the SLIC interface. Gain control is bypassed when setting the gain control parameters to zero. A low-latency HSS channel bypass with gain control is available.



A high-pass filter is applied to the input audio data from HSS interface in order to remove the unwanted low-frequency noise and safeguard the other algorithms from the harmful DC bias. The HPF uses a 3-dB cut-off frequency at 270 Hz in the narrowband mode or 150 Hz in the wideband mode. The HPF cannot be disabled until the Network Endpoint is stopped.

Echo cancellation is the most significant function in this component. EC cancels the echo generated by the hybrid of local telephone interface and phone set so that the other party connected to the channel will not hear the echo. In other words, the beneficiary of EC is the remote party. DSP solution supports Line Echo Canceller (LEC) as pluggable module. DSP solution does not provide Acoustic Echo Canceller as pluggable module.

EC performance is mainly affected by two parameters: tail length and delay compensation (that is, XPARMID_NET_ECTAIL and XPARMID_NET_DELAYCOMP). Depending on the hardware circuits and telephone set, the tail length of 4 to 8ms is usually good enough if the telephone set is directly connected to the unit.

Since EC is very computation intensive, the longer tail length results in higher CPU occupancy. Changing the parameter of EC tail length requires that the Network Endpoint component be reset (by sending XMSG_RESET message). The CPU occupancy is about doubled if the channel is configured to the wideband mode. The maximum tail length is limited to 64 ms for wideband and 128 ms for narrowband mode.

EC can be made the most effective if the reference signal is properly aligned with the delayed echo signal. That is the purpose of adjusting the parameter of delay compensation. The value of the parameter should be determined according to the customer's specific hardware platform.

You can use the XPARMID_NET_ECENABLE parameter to enable or disable EC. The parameter XPARMID_NET_ECFREEZE, used to disable adaptation on the EC algorithm, should only be used in debugging.

Lip-Sync Delay control is part of network endpoint resource component. APIs are present to control the delay from 0 to 1000 msec in step of 1ms both at transmit and receive side.

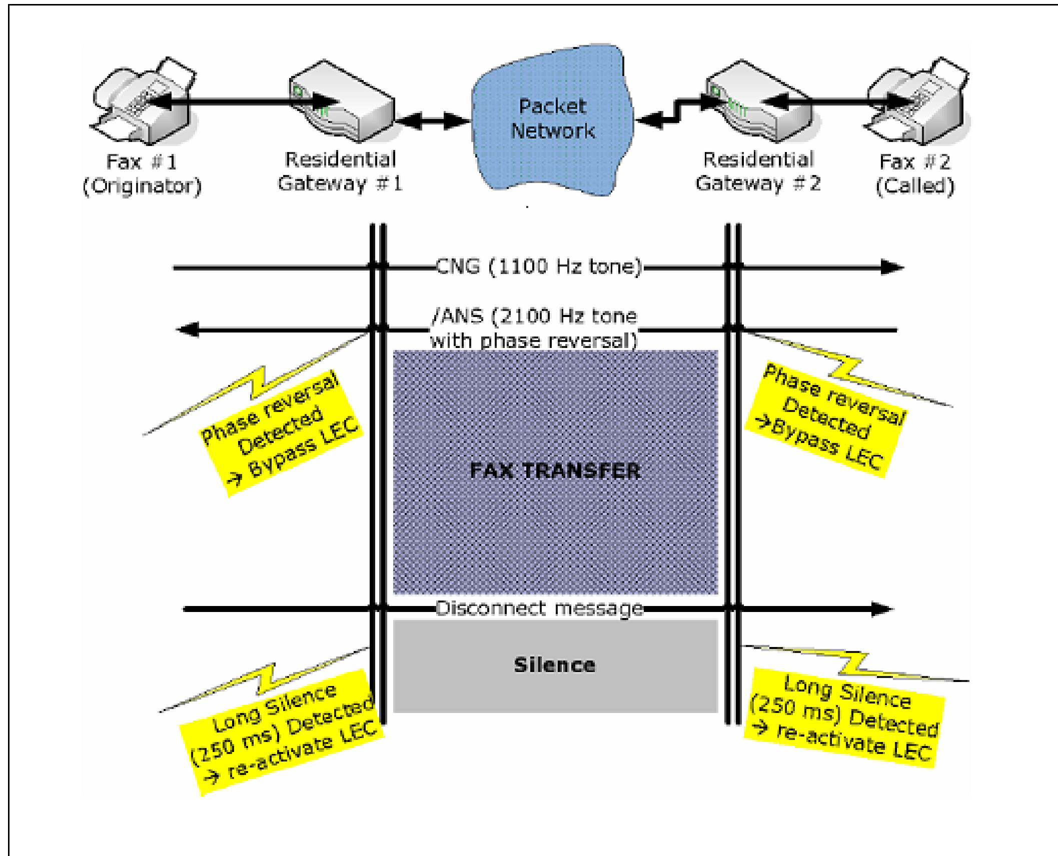
Network Endpoint resource also provides a complementary function of reporting hook state and detecting flash hook on behalf of the SLIC interface. The SLIC driver detects the hook state changes through the interrupt. The SLICs interrupt service routine can call the xFlashHookDetect() function, which reports the hook state via the XEVT_NET_HOOK_STATE event. The event data gives the hook state. If an on-hook followed by an off-hook transition within the time specified by the XPARMID_NET_FLASH_HK parameter, a flash-hook event will be reported.

Another complementary function is timer service. The user applications can set the timer counter via the XPARMID_NET_TIMER parameter. This counter is decremented by 1 each 10 ms. A XEVT_NET_TIMER event is generated when the counter is decremented to 0.

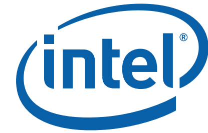
The Network Endpoint component is started with the default setup values automatically after initialization. The application can still start or stop it using XMSG_START or XMSG_STOP message for debug and test purpose. Stopping the component stops EC, HPF, and the complementary functions, but the audio data stream still continues and the A-law or µ-law conversion still functions in the narrowband mode. In other words, stopping the Network Endpoint component does not affect data transfer from/to the HSS interface.

The Network Endpoint component can be configured to report events related to fax over IP application, such as the detection of 2100 Hz tone with periodic (450+/-25 ms) phase-reversals, the detection of T30 preamble or the detection of user programmable silence threshold level and silence duration for fax operations. Figure below illustrates use case for Tone Disabler Events and events handling in typical fax over IP application.

Figure 7. Example of Tone Disabler Event and Event Handling in Fax over IP application



There are 4 modes for Tone Disabler in LEC by setting the parameter `XPARMID_NET_ECTONEDISABLERMODE`. In `OFF` mode, the Tone Disabler is off with default value set to 0. In `Manual` mode, the user application has to manually bypass or manually reactivate LEC in response to phase reversal and silence detected events. The user application needs to bypass LEC or make LEC inactive by setting `XPARMID_NET_ECBYPASS` to value 1 (ON) when an event `XEVT_NET_SIN_PHASEREV_YES` or `XEVT_NET_RIN_PHASEREV_YES` received. On the other hand, the user application needs to reactivate LEC to value 0 (OFF) after both `XEVT_NET_SIN_ECENABLED` and `XEVT_NET_RIN_ECENABLED` events received. In this mode, LEC must be bypassed before Fax Transfer starts and reactivated on completion of Fax transfer. Refer to example of Use Case in figure above for details. Bypassing LEC during silence or reactivating LEC during a fax transfer could lead to unexpected behavior due to improper timing of bypass/reactivation of LEC. In `Auto` mode, LEC is automatically bypassed when an event `XEVT_NET_SIN_PHASEREV_YES` or `XEVT_NET_RIN_PHASEREV_YES` received and LEC is automatically reactivated after both `XEVT_NET_SIN_ECENABLED` and `XEVT_NET_RIN_ECENABLED` events received. However these events will not be reported. In `Auto with Event Report` mode, LEC is automatically bypassed when an event `XEVT_NET_SIN_PHASEREV_YES` or `XEVT_NET_RIN_PHASEREV_YES` received and these events are reported. In `Auto`



with Event Report mode, LEC is reactivated when both XEVT_NET_SIN_ECENABLED and XEVT_NET_RIN_ECENABLED events received and these events are reported.

By default, during initial setup, all Tone Disabler and T.30 Preamble events are enabled. Event reporting will be disabled once the event is detected and reported. Setting XPARMID_NET_EVTRPTENABLE to value 6 will enable event reporting of all the events irrespective of their current status. User application also has the flexibility to enable individual event reporting. The current value of XPARMID_NET_EVTRPTENABLE shows only the recent value written, but does not show all the currently re-enabled event reports. It also doesn't disable previously re-enabled events through this API. For example, if you enable report of TDIS_PhaseReversal_TxEvt, followed by another command to enable reporting of TDIS_PhaseReversal_RxEvt, reporting will be enabled for both events (TDIS_PhaseReversal_TxEvt and TDIS_PhaseReversal_RxEvt) although XPARMID_NET_EVTPRTENABLE parameter shows only the reporting of the latest enabled (Enable TDIS_PhaseReversal_RxEvt).

T.30 preamble pattern detection can be enabled on Tx or Rx path by enabling XPARMID_NET_T30PREAMBDETENABLE. T.30 preamble pattern may be observed in ITU-T T.30 based fax sessions.

User application can specify threshold and duration for silence detection after a fax session. To qualify for an silence event, average amplitude level of the continuous silence and minimum duration of the silence must be configured.

4.2 Encoder

The primary function of this component is to encode and packetize the audio data from the HSS and then send to the IP interface. The audio codec supports G.711, G.726, G.722, and G.729ab on 10-ms frame size, G.729.1 on 20-ms framesize, G.723.1 and T.38 Fax on 30-ms frame size. iLBC is supported on both 20-ms and 30-ms frame size. Other features include Automatic Gain Control (AGC), Automatic Level Control (ALC), Voice Activity Detector (VAD), and Multiple Frame per Packet (MFPP). In the following paragraphs, the possible effect of these features on voice quality or system performance is briefly discussed.

This component works in the wideband mode when using G.722. While using G.729.1, this component may work in the wideband or narrowband depending on the SLIC configuration.

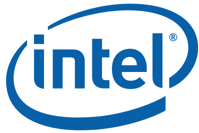
There are two automatic gain control elements: AGC in the egress side and ALC in the ingress side. Only one of these should be turned on, depending on what gain control functions are implemented in the remote party.

In the completed audio path when two parties are connected, enabling both AGC on one side and ALC on the other side may cause unexpected interaction and degrade voice quality. Typical VoIP equipment employs ALC, thus it is recommended that AGC is turned off and ALC is turned on (this is the default).

The VAD algorithm can distinguish active speech signal from the silence (background noise). During the silence period, the encoder only sends much smaller packets containing only the noise parameter at much lower rate. That helps to reduce network traffic.

Enabling VAD slightly impacts the voice quality.

Another effect of VAD is the change of average CPU occupancy. Enabling VAD in G.729ab and G.723.1 will significantly reduce the average occupancy because the most complicated processing of G.729ab encoder is eliminated during the silence and



background-noise period. However, VAD increases the CPU occupancy, when enabled with G.711, because the VAD algorithm is much more complicated than just the G.711 coder.

VAD is not available in G.726 and G.722.

Packing more frames into a packet (for example, MFPP) is another way to reduce network traffic. The application either specifies the number of frames per packet - in `XMSG_CODER_START` message when it starts the encoder - or modifies it - by setting the parameter `XPARMID_ENC_MFPP` at any time. Obviously, having MFPP increases the total latency and voice quality is more affected if the packet is lost. Typically, this trade-off of network traffic versus latency/voice quality is made depending on the target network and user preference.

You can query or change the coder type via the `XPARMID_ENC_CTYPE` parameter.

Switching the coder type on the fly may cause a few packets to be discarded. The number of frames per packet may be reduced automatically during switching if it exceeds the maximum allowed by the new coder type. If the encoder is started by `XMSG_START` message without specifying MFPP and the coder type, the current parameter values take effect.

G.726 has four different rates. Each of them is treated as a different coder type. They use dynamic RTP payload types that are negotiated by the call stack during call setup. The application is responsible for informing the DSP solution the payload type to be used in the current call by setting the payload type parameters. The parameters are:

- `XPARMID_ENC_G726_40_RTP_PLD`
- `XPARMID_ENC_G726_32_RTP_PLD`
- `XPARMID_ENC_G726_24_RTP_PLD`
- `XPARMID_ENC_G726_16_RTP_PLD`

Two packing formats are supported for G.726 of all the rates. One is described in RFC 3551 as commonly used for VoIP. Another is defined for ATM AAL in ITU-T I.366.2 Annex E. The `XPARMID_ENC_G726_PACK` parameter determines which format takes effect. Setting the parameter to `XPARM_G726_PACK_LSB` will choose RFC 3551 packing format or `XPARM_G726_PACK_MSB` for I.366.2 Annex E format.

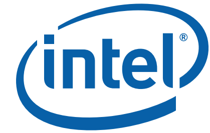
G.729.1 is a variable bitrate CODEC, and can support 12 bitrates (8, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32 Kbps). All the 12 bitrates are supported in narrow-band and wideband. These bitrates are dynamically changed based on bitrate negotiation between the two nodes. To run G.729.1 encoder in wideband mode, `XPARMID_ENC_G729_1_PCM_MODE` parameter should be set to `XPARM_G729_1_WIDEBAND`.

G.729.1 uses dynamic payload type, similar to G.726 codec payload type. The dynamic payload type parameter for G.729.1 encoder is:

- `XPARMID_ENC_G729_1_RTP_PLD`

iLBC, supports two different frame sizes. Each frame size is treated as a different coder type. It uses dynamic RTP payload type, that can be set during the call setup. The application is responsible for informing the DSP solution the payload type to be used in the current call by setting the payload type parameters. The parameters are:

- `XPARMID_ENC_ILBC_20MS_RTP_PLD`
- `XPARMID_ENC_ILBC_30MS_RTP_PLD`



The XPARMID_ENC_EVT_PKT message is used to set up the encoder to report bad packets. This is only intended for debugging since packet loss should not be monitored on an event basis.

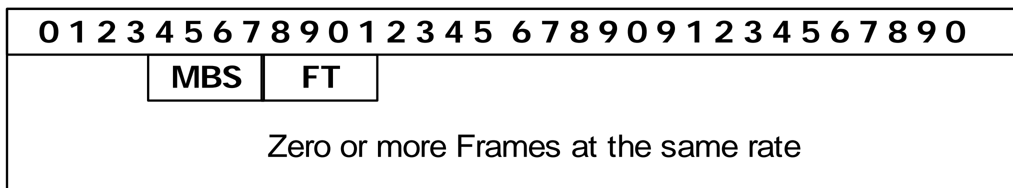
The user application starts Encoder by XMSG_CODER_START or XMSG_START message when a call is setup and stops it when the call is torn down. The Encoder is the component that enables data flow from HSS to the IP side.

A PASSTHRU CODEC type is provided for debugging purposes in the narrowband mode, in conjunction with the pass-through mode of the Network Endpoint component. When using PASSTHRU CODEC, no signal processing is done. The data in RTP G.711 packets are directly copied from HSS.

G.729.1 is an 8-32 Kbps Scalable Wideband (50 Hz -7000 Hz) speech and audio coding algorithm. It provides a standardized solution for a smooth transition from narrow-band to wide-band telephony. G.729.1 supports, 12 bitrates : 8000, 12000,14000, 16000, 18000, 20000, 22000, 24000, 26000, 28000, 30000, 32000 bits/sec.

The first layer at 8 Kbps, is called the core layer and is bitstream compatible with the ITU-T G.729/G.729.A coder. At 12 Kbps a second layer improves the narrow-band quality. Upper layers provide wideband audio between 14 and 32 Kbps with a 2 Kbps granularity. The codec operates on 20ms frames and the default sampling rate is 16 KHz. Input and output at 8 KHz is also supported at all bitrates.

Figure 8. G.729.1 RTP Payload Header



G.729.1 Payload consists of payload header of one Octet followed by zero or more consecutive audio frames, at the same bitrate. The payload header consists of two fields Maximum Bitrate Supported (MBS) and Frame Type (FT).

MBS value is used to limit the maximum bitrate that can be received by the decoder of the source node of the RTP payload. Decoder on the destination node receives the MBS value and updates XPARAMID_ENC_G729_1_MAX_RATE parameter of the destination Encoder. The value of the MBS field is set according to the following table:

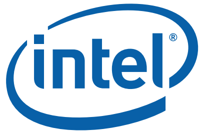


Table 1. MBS Table

MBS	Maximum Bitrate
0	8000
1	12000
2	14000
3	16000
4	18000
5	20000
6	22000
7	24000
8	26000
9	28000
10	30000
11	32000
12-14	Reserved
15	No_MBS

FT is used to communicate the frame encoded rate. Values 12 -14 are reserved values and the packets with reserved FT values are neglected. FT value 15 is used for 0 payload, and the payload is reduced to the payload header. This is used to communicate MBS value whenever there are no packets to send. FT field is used to indicate the frame size and the bitrate, as shown in the table:

Table 2. FT Table

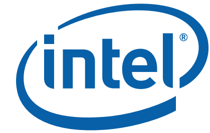
FT	Encoding Rate (bits/sec)	Frame Size (bytes)
0	8000	20
1	12000	30
2	14000	35
3	16000	40
4	18000	45
5	20000	50
6	22000	55
7	24000	60
8	26000	65
9	28000	70
10	30000	75
11	32000	80
12-14	Reserved	
15	NO_DATA	0

The communication between the two gateways happens by sending and reciving RTP payload over network, let us say Gatway A :Channel 1 and Gatway B:channel 2.

Initially, the the bitrate on both sending and receiving side on both the gateways is 32 Kbps. G.729.1 runtime bitrate negotiation through RTP payload header is shown in Figure 9

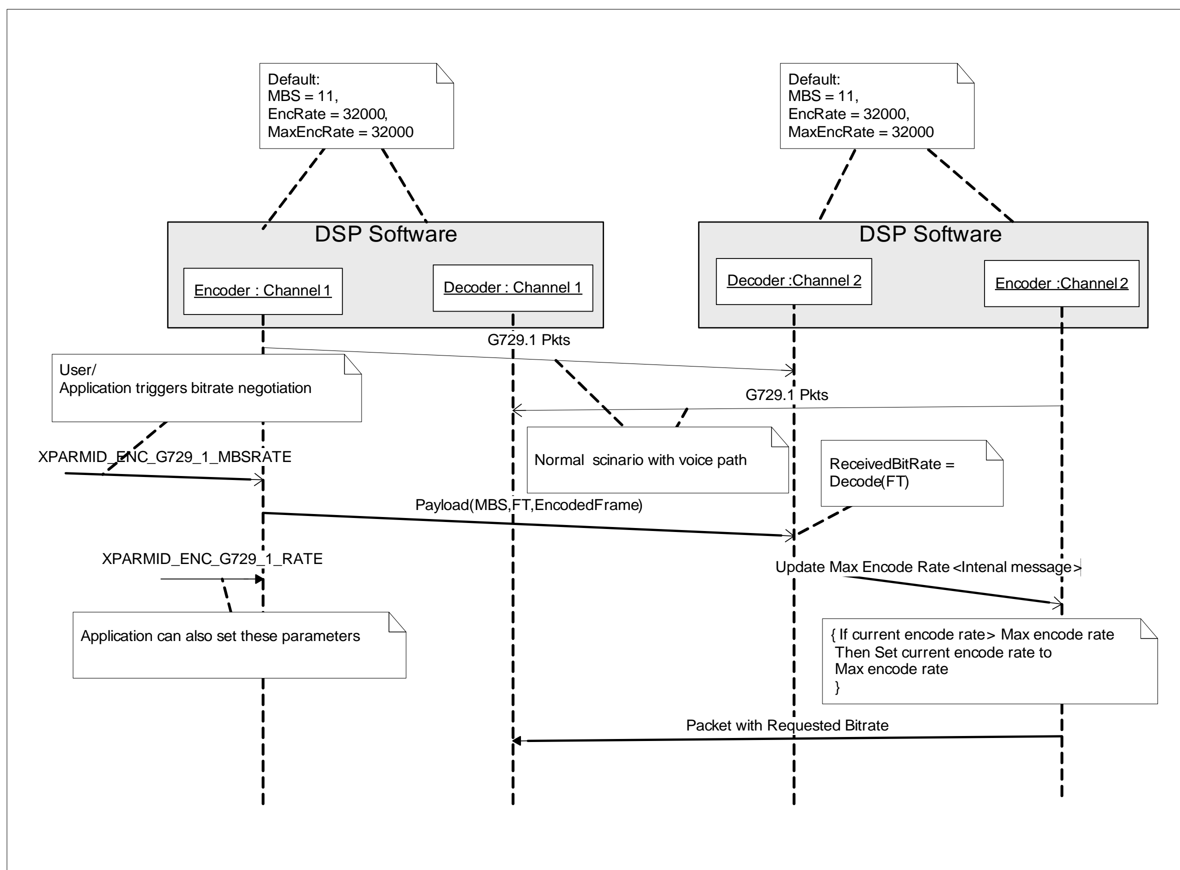
Case: If Gatway A :channel 1 receives voice data say at a maximum bitrate of 16 Kbps, the following sequence of negotiation steps happens:

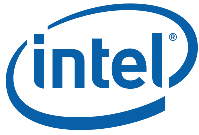
- Application inputs the `XPARAMID_ENC_G729_1_MBS` parameter channel 1 with 3, (MBS code for 16 Kbps) to the DSP software. Channel 1 encoder embeds the `XPARAMID_ENC_G729_1_MBS`, along with the FT code in the RTP payload header and delivers the payload.



- Channel 2 decoder receives the payload, and decodes the payload header. Channel 2 Decoder updates the `XPARAMID_ENC_G729_1_MAX_RATE` parameter of channel 2 Encoder by decoding the `XPARAMID_ENC_G729_1_MBS` parameter to 16 Kbps, that is received from channel 1.
- `XPARAMID_ENC_G729_1_MAX_RATE` is a read-only parameter; application can only read and it cannot modify this parameter value. This parameter is used to limit the current encoding rate.
- Application can set two parameters on encoder: `XPARAMID_ENC_G729_1_MBS` and `XPARAMID_ENC_G729_1_MAX_RATE`. Application sets these two parameters on the channel 2 encoder while `XPARAMID_ENC_G729_1_RATE` is made as the current encoding rate.
- Channel 2 encoder compares the current encoding rate with the `XPARAMID_ENC_G729_1_MAX_RATE` parameter. If the channel 2 encoding rate is greater than the `XPARAMID_ENC_G729_1_MAX_RATE` parameter, the current encoding rate is set equal to the `XPARAMID_ENC_G729_1_MAX_RATE` parameter.
- Channel 2 encodes the packet with 16 Kbps or a bitrate of `XPARAMID_ENC_G729_1_RATE` which is less than 16 Kbps.

Figure 9. G.729.1 Bitrate Negotiation through RTP Payload





4.3 Decoder

The Decoder receives the encoded audio packets from the IP interface and converts them to the audio stream to the HSS interface. Similar to the encoder, the decoder supports G.711, G.729ab, G.723.1, G.722, G.726, G.729.1 and iLBC coder types and additional features like Comfort Noise Generator (CNG), ALC, Packet Loss Concealment (PLC), and Jitter Buffer.

CNG is the counterpart of VAD in the encoder. For G.729ab and G.723 coders, CNG is built into the decoder algorithms and cannot be turned off. For G.711, disabling CNG will result in the pure silence between active speech periods if VAD is enabled in the remote party.

CNG is not available in G.726 and G.722.

The PLC algorithm uses the previous speech signal to repair the lost frames. But it cannot repair any big chunk of consecutive frames lost. Because of the complexity of PLC algorithm, it will increase the processor occupancy during packet loss when using G.711, G.726, and G.722 coders. But since they are relatively low computation coders, the resultant processor occupancy rates are still lower than that of G.729ab and G.723.

The PLC algorithm is always enabled.

The Decoder automatically handles MFPP if a received packet contains multiple frames. The application starts Decoder when a call is set up, using a `XMSG_CODER_START` message (`frmsPerPkt` field in the message is ignored for the Decoder). Currently, both the Encoder and Decoder support MFPP frame counts that are limited by internal buffer size.

The Jitter Buffer regulates the flow of data from the IP interface to the HSS interface. This is necessary since encoded audio packets from the IP interface are being transmitted on the IP network in real time using RTP protocol. This means packets can be delayed, out-of-order, duplicated, or lost without re-transmission. To perform this function, the Jitter Buffer delays incoming packets to allow delayed and out-of-order packets to arrive and be delivered to the HSS interface correctly. Depending on IP network conditions, this delay is dynamically adjusted by the Jitter Buffer.

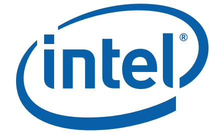
The Jitter Buffer monitors network conditions by checking the timestamps in the incoming DSP solution packets against the local clock. The correct sequencing of audio packets is also done with the help of the timestamp. The Jitter Buffer implements a proprietary delay profiling algorithm that (compared with the algorithm specified by RFC 3550) provides better tracking and improves voice quality.

There is typically a trade-off of delay versus being able to recover more delayed packets in real data networks. The Jitter Buffer allows the user application to balance this by two parameters:

- `XPARMID_DEC_JB_MAXDLY` - Specifies the maximum desired jitter delay in ms (current range is 10 to 500 ms)
- `XPARMID_DEC_JB_PLR` - Specifies the allowable packet loss rate in 0.1% units

The jitter buffer automatically determines the jitter delay based on the network delay profile it keeps from the desired packet loss rate, subject to the limit of the maximum allowed jitter delay parameter. By setting the allowable packet loss rate judiciously, a balance between voice quality and latency can be achieved in real network conditions.

If a packet has not arrived after the allowable jitter delay, the packet is declared lost and the Decoder is instructed to perform packet loss concealment. The Jitter Buffer also handles VAD packets and MFPP packets appropriately.



Since they can be at a different rate than the codec frame rate and the timestamps are event-based instead of frame-based, the Jitter Buffer handles RFC 2833 tone packets independently.

The Jitter Buffer is at the front-end of the ingress side. The user application uses the `xPacketReceive()` function to copy the encoded audio packets from the IP interface directly into the jitter buffer memory.

The Jitter Buffer also provides additional statistics such as maximum jitter, minimum jitter, mean jitter, standard deviation, jitter buffer maximum delay, jitter buffer absolute maximum delay, jitter buffer nominal delay and jitter discard rate. Some of the Jitter Buffer statistics variables, such as Residual Echo Return Loss (from Network Endpoint component) and Multiple-Frames-per-Packet (from Encoder component), are not directly related to Jitter Buffer or Decoder component.

Maximum jitter variable (`maxJitter`) is defined as a 32-bit variable. Suppose the packets received are 0, 1, 2, till N-th packet within the jitter buffer length. To calculate the `maxJitter`, find the maximum jitter of the packets 1 and 2. Store this in a variable `maxJitter`. Now, calculate the maximum jitter of packets 2 and 3 and compare this with the value in `maxJitter`. Store the higher of the two values in `maxJitter`. In this way, update the value of the `maxJitter` variable continuously. When you execute the command to get jitter statistics, you get the current value stored in `maxJitter`.

Minimum jitter variable (`minJitter`) is defined as a 32-bit variable. Suppose the packets received are 0, 1, 2, till N-th packet, to calculate the `minJitter`, find the minimum jitter of the packets 1 and 2. Store this in a variable `minJitter`. Now calculate the minimum jitter of packets 2 and 3 and compare this with the value in `minJitter`. Store the lower of the two values in `minJitter`. In this way, update the value of `minJitter` variable continuously. So, when you execute the command get jitter statistics, you get the current value stored in `minJitter`. The initial value for `minJitter` is maximum positive number and is equal to `0x7fffffff` in terms of time stamp units and while reporting it gets right shifted appropriately to give the value in milliseconds.

Mean jitter variable (`meanJitter`) is defined as a 32-bit variable. Suppose the packets received are 0, 1, 2, till N-th packets and jitters associated with packets from 1 to N are J1 to JN. We will get the `meanJitter` by using the formula given below:

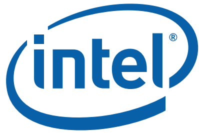
$$mean_jitter = \frac{1}{N}\sum_{i=1}^{N} J(i)$$

where J (i) is the instantaneous jitter and N is the number of packets accumulated

Standard deviation variable (`devJitter`) is defined as a 32-bit variable. Suppose the packets received are 0, 1, 2, till N-th packets and jitters associated with packets from 1 to N are J1 to JN. We will get the `devJitter` by using the formula given below:

$$S = \sqrt{\frac{1}{N}\sum_{i=1}^{N} (J(i) - mean_jitter)^2}$$

Where N is the number of packets accumulated, J(i) is the instantaneous jitter and S is standard deviation jitter.



Jitter buffer maximum delay variable (`jbMaxDelay`) is defined as a 16-bit variable. This is the current maximum jitter buffer delay in milliseconds which corresponds to the earliest arriving packet that would not be discarded. In simple queue implementations this may correspond to the nominal size. This parameter has the same value as `XPARMID_DEC_JB_MAXDLY`.

Jitter buffer absolute maximum delay variable (`jbAbsMaxDelay`) is defined as a 16-bit variable. This is the absolute maximum delay in milliseconds that the adaptive jitter buffer can reach under worst case conditions. If this value exceeds 65535 milliseconds, then this field conveys the value 65535. This parameter must be provided for adaptive jitter buffer implementations and its value must be set to JB maximum for fixed jitter buffer implementations. This value is 500ms at present for adaptive jitter buffer and 200ms for fixed jitter buffer.

Jitter buffer nominal delay variable (`jbNominalDelay`) is defined as a 16-bit variable. This has been assumed to be the actual jitter buffer delay length at a given instant.

Jitter discard rate variable (`discardRate`) is defined as an 8-bit variable. Discard rate is defined by the formula as given below. The maximum value is limited to 255 to avoid overflow and the integer part is considered.

```
discardRate (%) = [(Packets arrived early + Packets dropped due to
                   overflow + packets dropped due to underflow +
                   Packets arrived late)/(Total packets expected)]
                   * 256
```

Residual Echo Return Loss (RERL) variable (8 bits) and Multiple-Frames-per-Packet (MFPP) variable (8-bits) are defined along with other jitter buffer statistics. The residual echo return loss value may be measured directly by the VoIP end system's echo canceller or may be estimated by adding the echo return loss (ERL) and echo return loss enhancement (ERLE) values reported by the echo canceller. RERL is given by the formula mentioned below.

```
RERL(dB)= ERL hybrid(dB) + ERLE(dB)
```

ERLE is running echo attenuation by LEC, averaged over current and past speech frames. It is computed and updated only for single-talk with far-end speech.

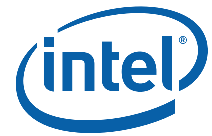
Multiple-Frames-per-Packet variable is set through user application in encoder component. In this case, there is no computation involved. This parameter is again given back to the user application, when the status of MFPP is enquired through codelets application's diagnostics menu.

The Jitter Buffer statistic can be reset through inbound message structure "`XMSG_GET_JBSTAT`". The example shown in Figure 10 depicts a scenario where the reset command is executed for resetting mean jitter. The assumption made here is that the jitter for a packet is already available. In this example, the resetting mechanism for mean jitter is explained.

- Suppose, at a given point, say at time t1, you want to get the `mean_jitter`. Assume that till that point, N packets have arrived. So, for N packets, the `mean_jitter` will be calculated. Now, say you execute reset command (at time t1), and want to calculate the mean at time t2. Suppose "K" packets have arrived by time t2. At this time, if you request for calculating mean you will get the result for the packets from "N+1" to "K".

Note: There is no reset applied at time t2.

- Now, suppose at time t3 you want to get `mean_jitter`. Assume that till time t3, L packets have arrived. Then in this case, you will get the `mean_jitter` for packets from "N+1" to "L".



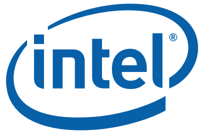
The above principle holds good for minimum jitter, maximum jitter, standard deviation and jitter discard rate as well.

Figure 10. Jitter Buffer Statistics Resetting Mechanism

0
1
2
3
4
Time = t1 N
N+1
Time = t2 K
Time = t3 L
Calculate the mean_jitter and apply reset
Calculate the mean_jitter
Calculate the mean_jitter

You can query the coder type via the `XPARMID_DEC_CTYPE` parameter. During decoding processing, the coder type may be switched automatically according to the received RTP payload type or changed by the user's application.

To allow automatic coder switch, set the `XPARMID_DEC_AUTOSW` parameter in which each bit represents a coder type. For instance, setting the parameter to (`XPARM_DEC_AUTOSW_G711MU | XPARM_DEC_AUTOSW_G711A`) allows the decoder to automatically switch between G.711 A-Law and μ-Law coder types. The received packets will be discarded if they do not match either of the two coder types.



Setting the parameter to XPARM_DEC_AUTOSW_OFF disables the auto-switch feature.

Setting the parameter to XPARM_DEC_AUTOSW_ALL enables Decoder to switch to all supported coder types.

You can also change the coder via the XPARMID_DEC_CTYPE parameter at any time. But keep in mind that the coder type may switch anyway if auto-switch is enabled. When the decoder is started by a XMSG_START message without specifying the coder type, the current parameter takes effect. Changing the coder type on the fly may cause a few packets to be lost.

The DSP solution reports the changes of received RTP payload type through the event message (XMSG_EVENT). The event code is XEVT_DEC_PACKET_CHNG. The event data 1 gives the coder type associated with the changed payload type and the event data 2 is the received RTP payload type. From the event and the setting of XPARMID_DEC_AUTOSW parameter, the user application can determine if the coder type is switched automatically or not.

For example, if the coder type reported by the event matches any of the ones set in the XPARMID_DEC_AUTOSW parameter, the event also indicates the decoder has switched its coder type accordingly. The event report can be enabled or disabled by the XPARMID_DEC_EVT_PKTCHNG parameter.

G.726 has four different rates. Each of them is treated as a different coder type. They use dynamic RTP payload types that are negotiated by the call stack during call setup. The application is responsible for informing the DSP solution of the payload type to be used in the current call by setting the payload type parameters. The parameters are:

- XPARMID_DEC_G726_40_RTP_PLD
- XPARMID_DEC_G726_32_RTP_PLD
- XPARMID_DEC_G726_24_RTP_PLD
- XPARMID_DEC_G726_16_RTP_PLD

Two packing formats are supported for G.726 of all the rates. One is described in RFC 3551 as commonly used for VoIP. Another is defined for ATM AAL in ITU-T I.366.2 Annex E. The XPARMID_DEC_G726_PACK parameter determines which format takes effect. Setting the parameter to XPARM_G726_PACK_LSB will choose RFC 3551 packing format or XPARM_G726_PACK_MSB for I.366.2 Annex E format.

G.729.1 uses dynamic payload type, similar to G.726 codec payload type. The dynamic payload type parameter for G.729.1 encoder is:

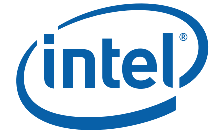
- XPARMID_ENC_G729_1_RTP_PLD

To run G.729.1 decoder in wideband mode, XPARMID_DEC_G729_1_PCM_MODE parameter should be set to XPARM_G729_1_WIDEBAND.

iLBC, supports two different frame sizes. Each frame size is treated as a different coder type. It uses dynamic RTP payload type, that can be set during the call setup. The application is responsible for informing the DSP solution the payload type to be used in the current call by setting the payload type parameters. The parameters are:

- XPARMID_DEC_ILBC_20MS_RTP_PLD
- XPARMID_DEC_ILBC_30MS_RTP_PLD

The XPARMID_DEC_EVT_PKT parameter is used to setup the decoder to report packet loss. This is only intended for debugging since packet loss should not be monitored on an event basis.



The user application starts the Decoder together with the Encoder when a call is set up and stops it when the call is torn down. The decoder is the component that enables data flow from the IP side to the HSS. A PASSTHRU CODEC type is provided for debugging purposes, in conjunction with the pass-through mode of the Network Endpoint component in the narrowband mode. When using PASSTHRU CODEC, no signal processing is done. The data payload in RTP packet is directly copied from Jitter Buffer to the HSS.

4.4 Tone Generator

The Tone Generator is capable to generate single- or dual-frequency tone and amplitude-modulated tone. It has a set of pre-defined tones. And user-defined tones can be added. Several tone segments can be combined as a single tone signal. This is very useful to generate some special call progress tones.

Internally, a tone is represented by a template that contains information such as tone ID, frequencies, amplitude, and cadence. Current supported tones can have one or two frequencies (DTMF), each with its amplitude information. Modulated tones are supported by specifying the carrier frequency/amplitude and modulating frequency/amplitude. Tones, (especially call progress tones), can have a cadence, that is, an 'on' duration, following by an 'off' duration, and a repeat pattern.

The Tone Generator is a narrowband resource and cannot produce the frequency higher than 4,000 Hz.

All the tone templates, including DTMF and call progress tones, are pre-defined. Since call progress tones are country-specific, the application has to set the country code during initialization, so that Tone Generator can select the correct template table accordingly.

Overall tone volume can be changed by the `XPARMIDTNGEN_VOL` parameter.

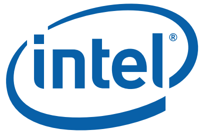
The application can play tones by sending an `XMSG_TG_PLAY` message with a list of tone IDs to be played sequentially. The definition of tone ID is compliant to RFC 2833 standard.

If tones are played while decoder has been started, the tone signal will overwrite or mix with the speech signal from the decoder according to the mode specified in the tone template. Most tones are of the overwrite-mode so that the speech is muted during the whole tone period. However, some tones have the cadence of a tone-on duration followed by a silent duration. For example, a call-progress tone, such as the call waiting notification tone, may require a short tone, followed by a long pause, and then the repetition of the tone-on/tone-off sequence. For these tones, the mix-mode is more appropriate, which allows the tone signal to be added to the speech so that the speech is not suppressed during the silence duration, or non-activated part of the tone.

If a continuous tone (for example, call-progress tone) is played, the user application can stop it by playing another tone or stop it explicitly using `XMSG_STOP` message.

The Tone Generator can also generate FSK modem signals compliant to ITU-V.23 or Bellcore* 202 specifications, depending on user mode selection via the `XPARMID_TNGEN_FSK_MOD` parameter. This is implemented for caller ID generation. To implement caller ID functionality, a user application has to directly control the SLIC telephone interface and implement the caller ID transmit sequence.

FSK parameters such as baud rate, channel seizure bits (CS) length, mark bits length, and postmark bits length can be modified by the `XPARMID_TNGEN_FSK_RATE`, `XPARMID_TNGEN_FSK_CS`, `XPARMID_TNGEN_FSK_MARK`, and `XPARMID_TNGEN_FSK_POSTMK` parameters, respectively.



The Tone Generator also generates the corresponding tones when RFC 2833 packets are received, if RFC 2833 tone generation is enabled by the `XPARMID_TNGEN_RFC2833` parameter. The RTP user application needs to classify the RFC-2833 packets based on the negotiated dynamic payload type, and encode the media field in the headers to indicate to the DSP solution that these are RFC-2833 packets. RFC-2833 tones will override audio frames if both are present.

Although the Tone Generator has a set of pre-defined tones including the DTMF tones and the call progress tones of the United States, Japan, and China, the user applications can add more tone definitions through the `xBuildToneTG()` function in which a new tone is defined by a list of tone segments and associated tone ID.

Each segment is specified by a set of parameters including the signal types (single or dual frequency or amplitude-modulated tone), amplitudes or modulation rate, on/off durations and numbers of repetitions. A total of 64 tone segments can be added. Since a tone can contain multiple segments, the number of tones that can be added can be less than 64. The multiple segment tones are typically necessary in the country-specific call progress tone definitions.

You can replace the pre-defined call progress tones with the newly added tones by specifying the same tone IDs.

The user-defined tones must be added during initialization time following `xDspSysInit()`.

4.5 Tone Detector

The Tone Detector is also a narrowband resource and is able to detect single- or dual-frequency tones with the frequency range from 300 to 3,500 Hz, using an FFT analyzer. Besides the pre-defined tones, you can add new criterion tables during initialization to detect user-specified tone signals. TD can receive FSK data.

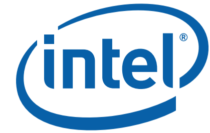
To reliably detect a dual tone, it is required that the frequencies of the dual-tone signal are separated by at least 200 Hz.

Internally all the tones to be detected (that is, DTMF tones) are described by a list of templates that contain the criteria of frequencies, energy, SNR, durations, and so on.

To use any features provided by the Tone Detector, the user application needs to first start Tone Detector by sending `XMSG_START` message. The basic function of Tone Detector is to report tone events that are enabled by setting the parameter `XPARMID_TD_RPT_EVENTS`. Tone-on and/or tone-off events are reported according to the parameter. Tone events are reported via the `XMSG_EVENT` message in which the event data 1 field indicates tone ID and event data 2 field is the time stamp in 10-ms units.

Instead of being notified by tone events, the user application may want to receive a DTMF digit string, for example, a telephone number entered from the telephone set. For this purpose, the user application can use the `XMSG_TD_RCV` message and specify number of digits it expects and the termination conditions. Tone Detector will return the result via `XMSG_TD_RCV_CMPLT` message once the digits are collected or the termination conditions are met.

One scenario of using this feature is call setup. For example when the application detects the off-hook state of the telephone, it plays the dial tone and then starts to collect 10 digits of calling number entered by the telephone. It waits for 20 seconds for the first entering. After that, it stops collecting the entering of the digits if getting all the 10 digits as expected, or no entering in 5 seconds after any digits, or any special digits (star or pound) entered, or the total time of 25 seconds passed before getting 10 digits.



In this case, the application use the XMSG_TD_RCV message, specifying all the termination conditions mentioned above in the message correspondingly. The XMSG_TD_RCV_CMPLT message returns the collected digits and tells the reasons why collecting the digits is stopped. If the tone event report is enabled during receiving digits, the first digit entering is also reported as an event. The application can use that event to stop the dial tone in the above example. Then the tone event report is temporarily disabled for the rest of digits automatically.

The Tone Detector can also receive and decode FSK signals used in Caller ID specifications. Currently it works for Bellcore* 202 or ITU-V.23 at a fixed, 1,200-bps baud rate. To start receiving FSK data, the application sends the XMSG_TD_RCV_FSK message and receives the XMSG_TD_RCV_FSK_CMPLT message with the decoded data once completed, or when the specified timeout has expired.

During receiving FSK, all other tone detection features are temporally suspended.

Another feature of the Tone Detector is tone clamping. The Tone Detector mutes the input audio stream from HSS during the period when a tone signal is detected. For VoIP applications, this feature is primarily used to implement out-band DTMF, because the tone signal is often distorted by speech coder like G.729ab. Since it takes about 30 ms to detect a tone, up to 30-ms tone signal may already leak out before it is clamped. To prevent tone leakage, the user application can enable the look-ahead buffer by setting the buffer size parameter XPARMID_TD_TC_FRAMES to 1, 2 or 3 (in 10-ms units). Remember that enabling the look-ahead buffer increases the latency accordingly.

If RFC 2833 is enabled (XPARMID_TD_RFC2833E_ENABLE), the Tone Detector will generate RFC-2833 payloads for transmission from the user RTP application, via the registered RTP transmit function (using xDspSysInit). The RTP payload type for the RFC-2833 packets is specified via the XPARMID_TD_RFC2833E_PAYLOADTYPE message. The marker bit in the packet header is also set by the DSP solution.

The rate for RFC-2833 packet generation can be set by the user application (XPARMID_TD_RFC2833E_UPDATERATE, typical rate is either 50 ms or coder frame rate). The number of beginning-of-tone (XPARMID_TD_RFC2833E_NUMBOE) and end-of-tone (XPARMID_TD_RFC2833E_NUMEOE) redundant packet transmission can also be set by the user application.

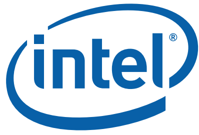
Normally, audio RTP packets are not transmitted during tones, but they can be enabled by turning off audio suppression (XPARMID_TD_RFC2833E_AUDIOSUPRESS).

Besides a set of built-in criteria to detect the DTMF tones, you can add new criterion tables, using xBuildToneTD(), to detect user-specified tone signals. Currently you can add the new tone detection ability for single or dual frequency tones but not amplitude modulated (AM) tones. The user-specified tone will be reported via the XMSG_EVENT message along with the tone ID and time stamp.

You cannot replace the pre-defined tone detection criteria. New tones are always added in addition.

4.6 Audio Player

The Audio Player component resource plays back the pre-recorded audio data to TDM and/or IP terminations. The Audio Player is designed to play cached voice prompts, that is, the audio data must be all pre-loaded into memory. The user application registers the audio data with the DSP solution via xDspRegCachePrompt() and obtains the prompt handles. Each handle represents a piece of audio data stored in contiguous memory.



Currently up to 32 handles can be registered permanently. The audio data must be recorded in G.711 A-law/µ-law or G.729 format and loaded into the memory as raw data format without any extra embedded information such as header, time stamps or other information.

The demo source code included in this release gives the examples of using hard coded audio data and loading the audio data from wave format files.

During playback, the application can play any selected data segments by specifying the handle, offset and length. This segment information must be supplied with the `XMSG_PLY_START` message. Each message can carry up to 14 segments which can be played back in any given order once or repeatedly.

The number of player instances in the DSP solution is configurable at initialization time. Each player instance has a dedicated location of the output audio stream where the encoded audio data is converted. To play back to an HSS or IP channel, the Network Endpoint resource or the Decoder resource has to listen to a player instance by connecting its input to the player. For details of audio stream routing, see Figure 12.

If an application uses the player resource only for playing on-hold music, one player instance is enough for the purpose since all the channels can listen to the same player. Otherwise each channel may need a dedicated player instance.

4.7 Audio Mixer

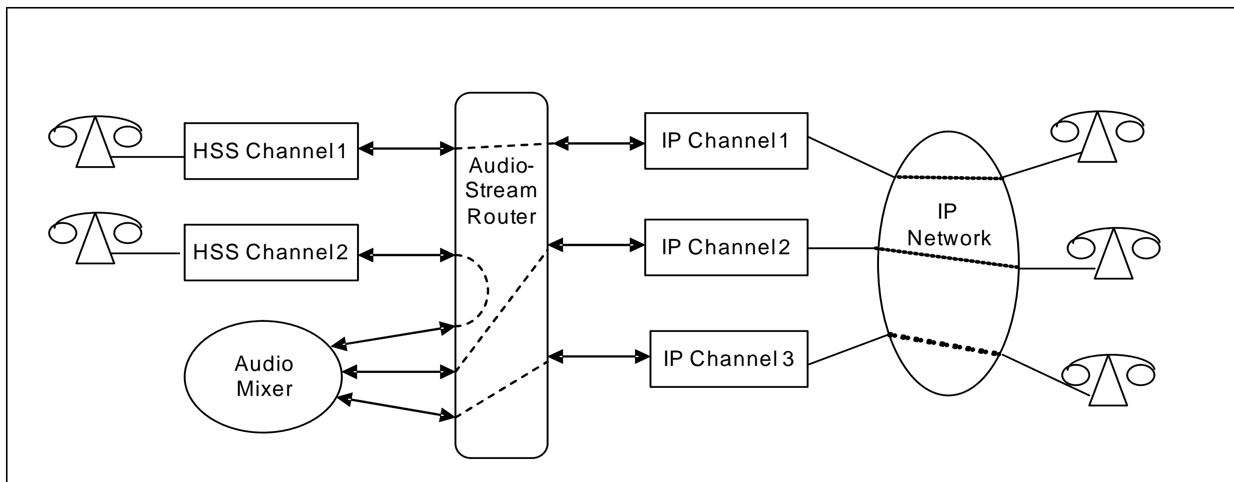
Audio Mixer mixes a number of audio streams to form an audio conference. The Mixer resource in the DSP solution is primarily used for three-way call applications. It does not have the pre-processing functions that are found in the audio conference resources such as active talker selection, and volume balance. Therefore, mixing too many parties may results in voice quality problems like background noise built up, unbalanced volumes on different parties when the network condition is not good.

DSP Solution provides four mixer instances. As one mixer instance is required to support one conference call, audio mixer component is enhanced to support up to four conference calls simultaneously, with up to five parties per conference call. However, actual number of Conference calls (mixer instances) and number of parties per conference call (TDM + IP terminations) possible may be less than 4x5, due to CPU load. For DSP Solution, up to four mixers enabled simultaneously with three parties per mixer is validated.

Note: The CPU cycles requirement varies from one speech codec to another. For example, running four conference calls with three parties per conference call using the codec G.711 on all IP terminations is possible on 533MHz CPU, but not with codec G.729ab. The codelet includes demo for four conference calls with three parties per conference call. Refer to Codelets Demo Guide document and codelets source code for example to configure multiple mixer instances.

Figure 11 shows how the audio streams are connected when a normal two-way and a three-way call are set up simultaneously. We can see during the three-way call there is no longer 1:1 association between HSS channels and IP channels and a mechanism of dynamically routing the audio streams is required. This will be discussed in the next section. Also we may need more IP channels than HSS channels if two parties of the three-way call come from IP side.

Figure 11. Audio Stream Connections in a Three-Way Call



The Mixer has multiple ports (pairs of input and output audio streams). Each port is to be connected to the resource (or party) that joins the call. The output of a port is the summation of all the inputs except for itself.

For example, consider three-party mixing:

- First party with input port L1 and output port T1.
  — The output of first party on port T1 is sum of data of input ports L2 and L3.
- Second party with input port L2 and output port T2.
  — The output of second party on port T2 is sum of data of input ports L1 and L3.
- The third party with input port L3 and output port T3.
  — The output of third party on port T3 is sum of data of input ports L1 and L2.

The Mixer resource is started and stopped by the `XMSG_START` and `XMSG_STOP` messages. It has the parameters that are used to link its audio input and output to other resources.

Currently, the Mixer operates only in the narrowband mode. The wideband audio data is converted to narrowband if a wideband channel is connected to the Mixer.

4.8 Audio Stream Router

The three-way call is an example that requires the audio streams be routed among the resources. Other examples are call transfer and IP tone detection.

To route the audio streams, we first break the DSP resources along the data path into a TDM termination and an IP termination which are connected by the router in between as shown in Figure 2.

The TDM termination contains the Network Endpoint resource.

The IP termination contains a set of resources (Decoder, Encoder, Tone Detector, and Tone Generator).

The TDM termination has a talk-port (T-Port) that supplies data to the router and a listen-port (L-Port) that receives the data from the router.

The IP termination has one T-Port shared by Decoder and Tone Generator and two L-Ports for Encoder and Tone Detector separately.

In general, a resource that generates PCM audio data has a T-Port as its output and a resource that receives the audio has an L-Port as its input. For example, an Audio Player instance has only one T-Port and a Mixer has multiple pairs of T-Ports and L-Ports.

The Router applies sampling rate conversion (SRC) automatically if the resources being connected are in different modes (wideband or narrowband).

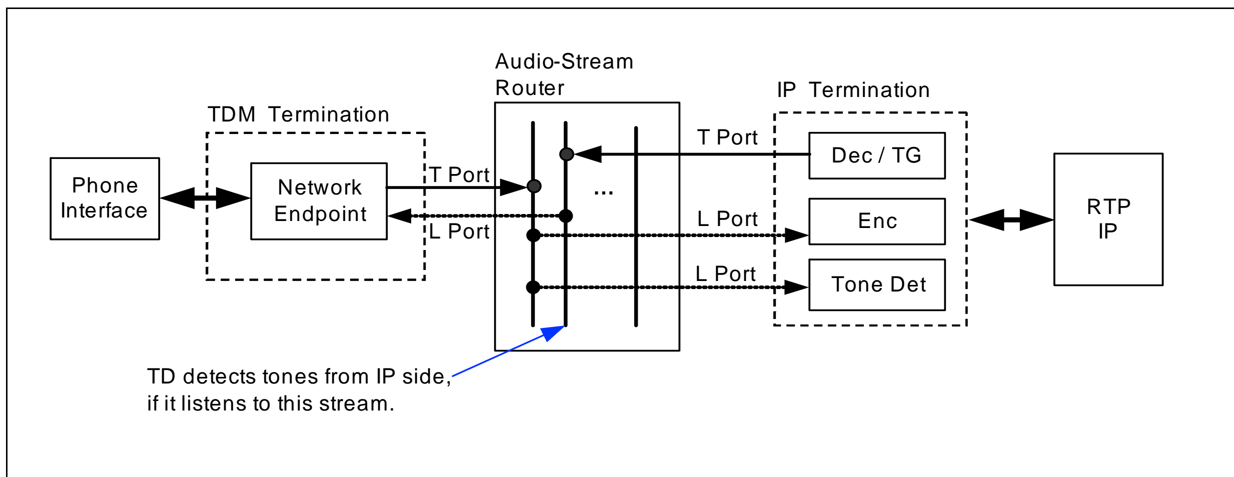
The DSP solution implements a distributed switch method to route the audio streams. The Audio Stream Router is not a control entity but a set of streams that can link the T-Ports and L-Ports.

All the T-Ports of the resources are assigned the dedicated streams permanently.

Routing is done by enabling an L-Port of a resource to listen to any streams by setting a parameter to the resource. In this way any T-Ports can be linked to any L-Ports.

Figure 12 shows a full-duplex connection between a TDM termination and an IP termination. In this figure, if the L-Port of the Tone Detector listens to the stream of the T-Port of the same IP termination instead of the one of TDM termination, then it will detect tones coming from the remote IP side.

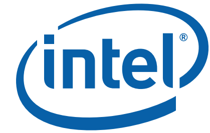
Figure 12. Terminations and Router



Each stream is specified by a unique ID number from 0. A null stream is given the ID as (-1). Any L-Ports listen to the null stream receive silence.

To make a connection between two resources, you should know what stream IDs are assigned to the T-Ports of the resources. Such information is available by calling `xDspGetResConfig()`. The function returns the base stream IDs for the T-Port for each type of terminations and resources (the TDM and IP terminations, Player and Mixer).

For example, the base stream ID of the TDM termination means the stream ID assigned to the T-Port of the first TDM termination channel. The T-Port stream of n-th channel (n=1, 2) is calculated as (base stream + n - 1). The base stream of the Mixer means the output stream of the Mixer's first port. The Mixer has 3 to 5 L-Ports that it mixes and it has the same numbers of T-Ports where the outputs of the mixes are transmitted.



Having the stream ID information for the T-Ports, you can have a resource listen to a particular T-Port by setting the L-Port stream parameter of the resource. For example, to detect the tone from IP side in the channel 2 of the IP termination, first obtains the base stream ID of the IP termination (suppose it is 4), Then the T-Port stream ID of IP termination channel 2 is 5 (4 + 2 - 1). Set the `XPARMID_TD_LP_STREAM` parameter of the Tone Detector to 5.

Network Endpoint and Encoder have their L-Port stream parameters too.

The `XPARMID_MIX_LP_STREAM` is such parameter ID of the first port of the mixer. For the rest of the ports, parameter IDs increases by 1 sequentially.

Examples of high-level message interfaces that link the terminations and the Mixer are also provided using the Message Agent approach.

In some applications, you may want to link two TDM terminations without IP involved (also called TDM switch or TDM bypass).

There are two modes for such connection. In the normal mode - when the `XPARMID_NET_HSS_BYPASS` parameter in Network Endpoint resource is set to `XPARM_OFF(0)` - the echo cancellation and front end gain control are applied to the audio path. This achieves a latency of approximately 25 ms and is a bypass at the Intel XScale® processor level.

In the short bypass mode when the parameter is set to `XPARM_ON(1)`, the connection is made within the NPE between the corresponding time slots, therefore the latency is reduced significantly to approximately less than 2 ms. In this mode, only the gain control remains in effect.

The short bypass can only be enabled if both TDM terminations to be linked are in narrowband mode or the audio data will be corrupted. To enable short bypass feature, initialize the DSP solution using the built-in HSS interface with at least eight active timeslots and register the necessary switching functions of the DSP solution patch.

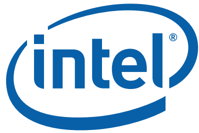
4.9 T.38 Fax

The T.38 Fax serves as the real-time fax gateway, between G3 fax machines and the IP network. Unlike the fax bypass mode in which the modulated fax data are directly packed in G.711 format and transmitted over RTP packets, the T.38 component transfers the demodulated T.30 commands and fax image data over UDP or TCP packets.

T.38 component contains three modules:

- A fax modem that establishes the T.30 session between the fax gateway and the local fax machine.
- T.38 CODEC that encapsulates the demodulated T.30 commands and HDLC data together with redundancy or forward error correction, into fax data packets suitable for transmission over UDP or TCP protocols.
- Packet Loss Recovery (PLR) that recovers lost packets from the redundancy or forward error correction on the receive side

The T.38 component is implemented as a separate entity from the voice resources (the Encoder, Decoder, Tone Detector, and Generator). A T.38 session is established by sending the `XMSG_T38_START` message. The T.38 resource returns the `XMSG_T38_CMPLT` message if the session is terminated. The termination reason can be retrieved from this message.



The T.38 component is mutually exclusive with voice resource components within the same channel during the run-time. It is the user applications' responsibility to stop the voice resources and start the T.38 component when switching over from voice mode to T.38 fax mode.

The included DSP codelet source code provides examples of how this can be accomplished in the VoIP gateway demonstration.

The DSP solution uses the same packet format to exchange voice and T.38 packets with the user applications. The media type field in the packet header indicates the packet types. In the TDM side, the fax modem uses the same PCM stream IDs assigned to the Encoder and Decoder with the same instance number to receive or generate the modulated fax data.

There are different modes that T.38 can operate in: in UDP or TCP mode, specified by the parameter `XPARMID_T38_TRANSPORT`, with packet redundancy or FEC (Forward Error Correction), specified by the parameter `XPARMID_T38_FEC`.

For TCP mode (currently not supported), the fax payload is transmitted via TCP/IP protocol. Packet loss in the network is recovered by retransmission via the TCP/IP protocol.

Encapsulation of the UDP or TCP packets is the responsibility of the user application. In UDP mode, the DSP solution emits the formatted UDPTL packet; in TCP mode, it emits the raw fax payload. The media type field in the DSP packet header identifies the type of packet being transmitted or received.

The `XPARMID_T38_RATE_NEG` parameter determines whether the rate negotiation is performed locally or remotely. Rate negotiation is typically done remotely for UDP mode, since the network conditions affect rate selection. Rate negotiation is typically done locally for TCP mode (currently not supported). In this case, `XPARMID_T38_TCF_THRSHLD` determines the error level threshold used to locally determine the rate.

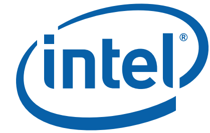
In UDP mode, T.38 specifies either packet redundancy or FEC for error recovery. For packet redundancy, the `XPARMID_T38_REDUNDANCY` parameter specifies the level of redundancy. This is only an indication of the overall level of redundancy. The actual redundancy in the payload is also determined by the type of fax payload (for example, signaling or image data). For FEC mode, the `XPARMID_T38_REDUNDANCY` parameter specifies the number of FEC messages per UDPTL packet.

The `XPARMID_T38_MODE` parameter specifies which variation of the T.38 protocol is to be used. The options are ITU T.38 or China T.38. In China Telecom mode, the `XPARMID_T38_DISCONNECT` parameter specifies whether the optional disconnect IP message is generated or not.

The `XPARM_ID_T38_ELLIPSIS` parameter is used to enable support of ellipsis added to Internet Fax Protocol in T.38 Corrigendum 1 (2001). The ellipsis is an extension marker in the protocol. When this extension marker was added in Corrigendum 1 of the T.38 recommendation, compatibility with the original T.38 recommendation was broken. At the time of writing most devices support the original version of the protocol and some newer devices support the Corrigendum version or are configurable to support both. This parameter should be set ON if it is known that the fax transaction will occur with a device that is configured for the Corrigendum to the recommendation.

4.10 Message Agent

The DSP solution exposes the individual media-processing resources and provides a basic set of message interface to user applications. This allows the maximal flexibility, but may not be convenient to the application development.



For example, the user application may have a state machine driven by the asynchronous events from the call stack and user inputs of the telephone set. For each event, the application has to send several control messages to the resource components and handle the replies. The large number of messages and their replies make the state machine more complicated. You may prefer one comprehensive message for each event. Such a message could include all the resource components involved.

The Message Agent can be viewed as a macro or scripting facility that allows multiple basic messages to be executed by one user message command. By eliminating multiple messages being passed between the DSP solution and user application, the associated context swaps are removed and operating efficiency gained. By providing a base of helpful pre-defined user messages, which can be modified and expanded, the integration between user application and the DSP solution can be expedited.

If you are going to replace an existing DSP solution with the DSP solution, you may have to modify you applications significantly because of the differences in the interfaces, or you may implement a translation layer to convert the interface. To build such a layer on top of the DSP solution may introduce extra overhead and inefficiency. With the Message Agent, you can embed such translation layer inside the DSP solution much more easily and efficiently because the message traffic is greatly reduced.

The Message Agent is a special resource component that does not have any media processing functions. To support the user-defined, high-level messages, supply a message decoder function registered with the Message Agent. The function decomposes the user message into a series of original control messages. The Message Agent will directly execute the control to resources based on the decoded message sequence. During the procedure, the responses from the resources are redirected to another user-supplied message encoder function, which composes the responses into one user-defined reply message sent back to the user application by the Message Agent. The only responses which are directly the results of the control messages such as `XMSG_ACK` and `XMSG_ERROR` are redirected. The messages that are the results of media data processing like `XMSG_EVENT` and `XMSG_TG_PLAY_CMPLT` are still sent to the applications as usual.

The Message Agent is enabled if a message decoder function is registered during the initialization via the xDspSysInit() function. The message encoder function is optional. If not registered, the replies from the resources are always sent to the application as usual.

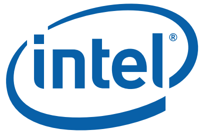
As examples, this release includes a set of high-level messages and the source code of message decoder and encoder functions. You can further extend and modify that message interface.

5.0 Programming Guide

This section discusses the rules and guidelines that should be followed when building user applications on top of the DSP solution.

5.1 Initialization

As the DSP solution is a standalone module or a layer of media processing, it must be configured and initialized properly before the application can interact with it. Initialization of HSS is done through the API `IxHssDriverHssPortInit (hss_port_config)`. Configuration details are provided in the `Hss_config` structure. For more details on this structure, refer to the *Intel® Infrastructure DSP Solution Version 1.2 API Reference Manual*.



The `Hss_config` structure includes:

- The Signal formats and time slot assignment on the HSS TDM bus are defined by the data structures `IxHssAccConfigParams` and `IxHssAccTdmSlotUsage`. This is given in `HSSConfig.h` file of the HSS driver.

Depending on the mode, each instance of the Network Endpoint resource must be linked to one or four time slots. If more time slots are activated, the data from the extra time slots are ignored and the data to those time slots are undetermined.

The link between the effective time slots and the instances of the Network Endpoint is specified by the `XDSPChanTdmSlots_t` data structure. If not given, all the instance of Network Endpoint will be configured to the narrowband mode and the first N time slots will be linked to the total N instance of Network Endpoint component sequentially. (For more details, see the *Intel® Infrastructure DSP Solution Version 1.2 API Reference Manual*.)

In the current release, the number of active time slots must be at least eight if the low-latency TDM switching feature is required. (The latency of HSS NPE will be minimized if eight or more time slots are enabled).

- The maximum number of instances is eight (except for Mixer, which has only four instances). The default number of eight will be used if an invalid number is given. For 533-MHz processor, the maximum number of channels that can be supported with the CPU occupancy under 50% is eight if only the G.711 coder is used or two if any other coders are used. For a 266-MHz processor, the channel density should be reduced by half. It is not recommended to have the higher channel density that leads to CPU occupancy of above 50%.
- Country code which determines the call-progress tone definitions and some the default FSK parameters.
- The base priority for the internal real-time data-processing and control tasks.
- The callback functions.

With user-supplied configuration information, the initialization follows these steps:

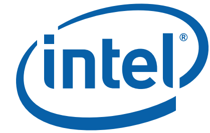
1. Download HSS NPE code and initialize HSS dependents.

(For more information, refer to the example shown in the demo source code.)

2. Add user-specified tone detection criterion tables to Tone Detector using `xBuildToneTD()`.
3. Call `xDspSysInit()` with the configuration information as described above. An assertion occurs if fatal errors happen (for example, if the memory is exhausted).
4. Add user-defined tone definitions to Tone Generator using `xBuildToneTG()`.
5. Use `xDspGetResConfig()` to retrieve the base stream information assigned to the different resource components. Such information is required when routing the audio streams between the resources. Also the function returns the actual resource configuration that can be different from what a user may have incorrectly specified.

5.2 Programming Model

A VoIP gateway application may contain several modules such as user interface, IP call stacks, and the DSP solution. The key functionality of the gateway application is to handle the call progress procedure: establishing calls and connecting the audio data path between two remote and local parties, then dropping the calls and disconnecting the data path accordingly.



From control point of view, this procedure can be characterized as the interactions among the DSP solution, IP call stack, and SLIC driver through asynchronous messages and commands. Such control logic is best implemented by a message-driven state machine model. The DSP solution's control interface is suitably designed to support this programming model.

To use the state machine approach, it is recommended that the user application spawn a dedicated task to handle the call-progress procedures. The DSP solution allows you to use the DSP solution release output message queue via the `xMsgWrite()` function.

You can use this function to send external messages (such as SLIC driver events or IP call stack messages) back to the user application to allow all message inputs to be consolidated. (In Linux*, this can be done in the client driver module). Then the user control task is pending on the message queue, using `xMsgReceive()`, to handle all the call progress-related messages from all these modules.

Figure 13 shows a general approach of such state machine model. In this programming model, a call progress scenario is represented by a sequence of states. Each state is characterized by:

- The actions it takes
- The messages it expects
- The next state it goes to

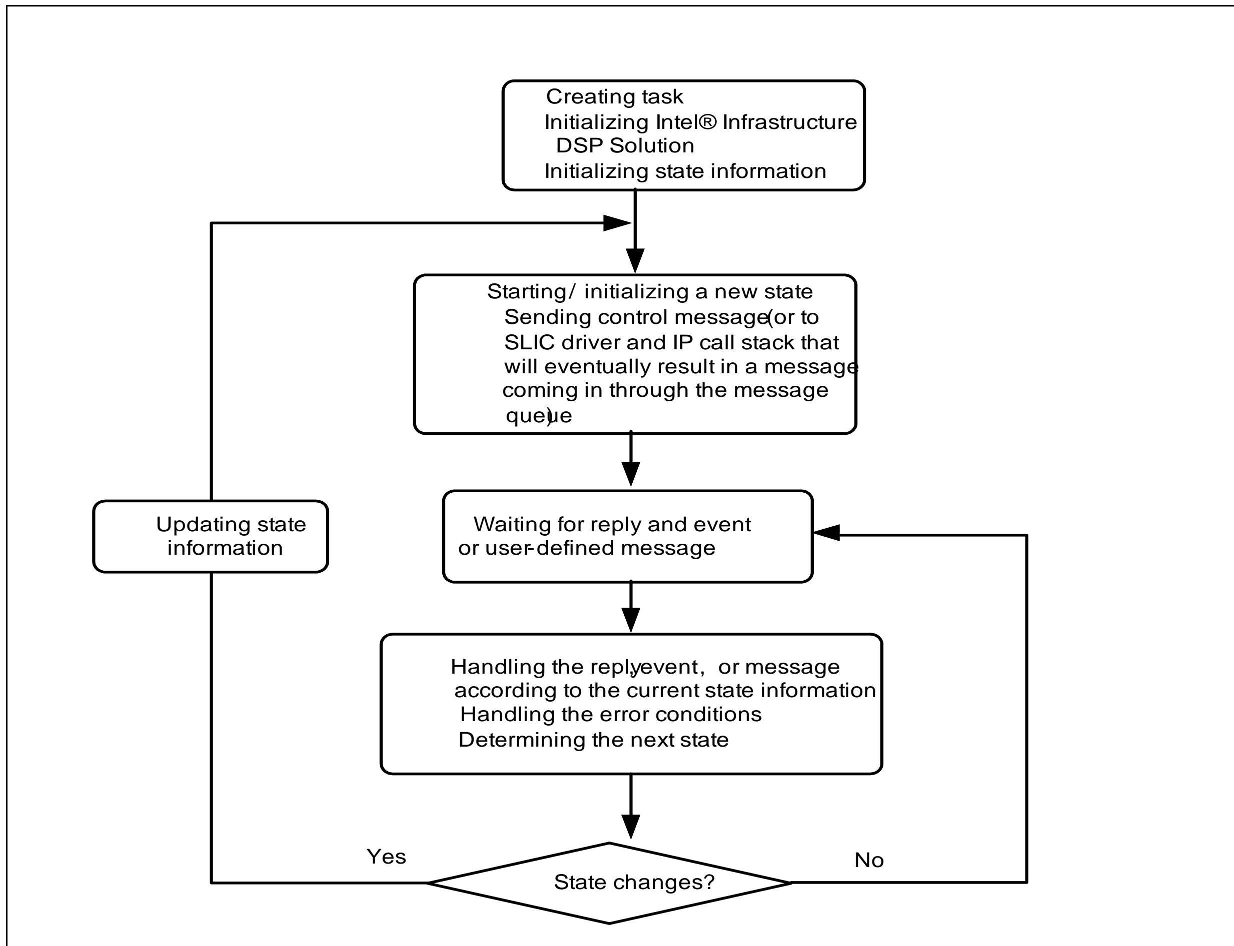
For example, the scenario of accepting a remote call can be represented by the following states:

- Idle State - Waiting for call-setup message from IP call stack.
- Ring State - Ringing the local telephone set and waiting for an off-hook event.
- Channel Setup State - Sending control messages to the DSP solution to start encoder, decoder, and tone-detector resources and waiting for the acknowledgements.
- Connected State - Acknowledging IP call stack that a local channel has been set up. Waiting for disconnect message from the call stack or on-hook event from the local telephone set.
- Teardown State - Sending control messages to the DSP solution to stop the resources and waiting for acknowledgements. Acknowledging IP call stack that the channel has been teardown. Going back to Idle State.

The actual state machine will be more complicated when taking all the possible error conditions into account. For instance, timeout message must be handled in Ring State if the call is not answered.

The major advantage of such a programming model is high efficiency and performance. In Linux, it also helps the DSP solution maintain its real-time behavior. The Gateway Demo included in this release is a good example of this programming model.

Figure 13. General State-Machine Approach for Client Applications



6.0 OS-Specific Issues

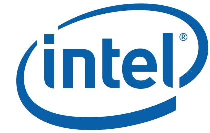
You should understand some OS-specific issues to design the overall software appropriately. Since the DSP solution only supports Linux OS, this section only covers Linux.

6.1 Linux*

If target-product cost is a major consideration, Linux will likely be the choice for use with the DSP solution. This OS will require some extra development effort and caution because:

- The DSP solution in Linux is fully in user mode. The software creates the user mode threads shown in Table 3 and Table 4.

You can configure the realtime task (taskPriReal) priority, control task (taskPriCtrl) priority and PCM read task (taskPriPCMRead) priority of dspcfg structure of Intel® Infrastructure DSP Solution Version 1.2. For better performance, realtime task and PCM



read task should have higher priority than any other tasks. The task thread priority level ranges 1 (lowest) to 99 (highest). All tasks are scheduled in Round-Robin with Linux* Kernel Scheduler.

Table 3. Linux* User Mode DSP Library Threads

Thread Name	Priority	Description
DspCtrlTsk	IX_DSP_CODELET_CTRLTASK_PRI (Default Value=10)	Real-time task. Control thread. Pending on in-bound message queue. Triggered by incoming control messages.
DspRtTsk30	IX_DSP_CODELET_REALTASK_PRI (Default Value=15)	Real-time task. Wakes up every 30 ms synchronously with PCM data. Executes G.723 CODEC algorithms, fax modem and T.38 CODEC algorithms. and iLBC CODEC algorithms (for 30-ms frame size)
DspRtTsk20	IX_DSP_CODELET_REALTASK_PRI (Default Value=15)	Real-time task. Wakes up every 20 ms synchronously with PCM data. Executes G.729.1 CODEC algorithms, and iLBC CODEC algorithms (for 20-ms frame size)
DspRtTsk10	IX_DSP_CODELET_REALTASK_PRI (Default Value=15)	Real-time task. Wakes up every 10 ms synchronously with PCM data. Executes all the DSP algorithms supported in the current release.
DsrPcmDriverRead	IX_DSP_CODELET_PCMREAD_TASK_PRI (Default Value=15)	Real-time task. Reads PCM data from HSS driver

Table 4. Linux* User Mode DSP Application Threads

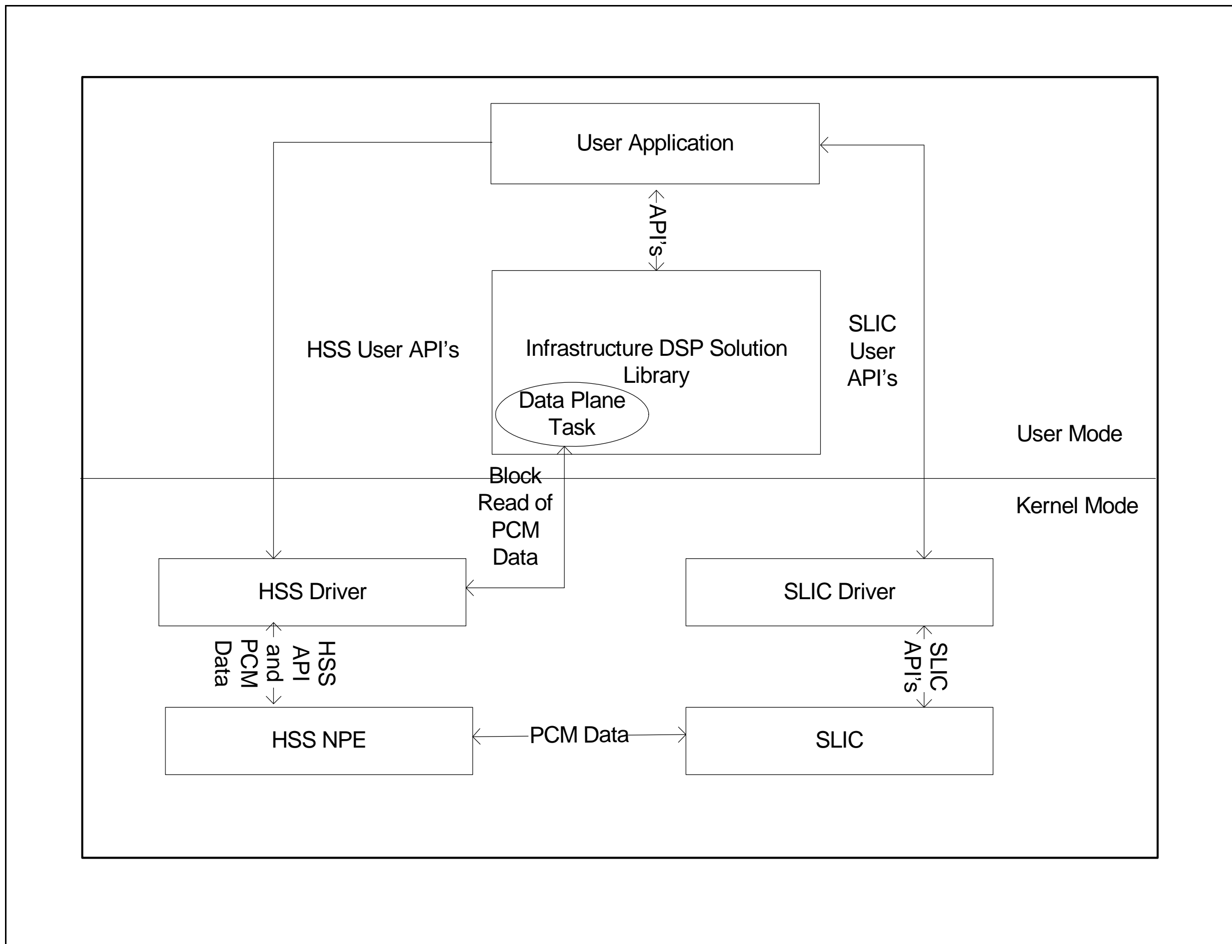
Thread Name	Priority	Description
cbthread	Default Value = 40	Real-time task. Thread monitors SLIC events.
Socket thread	Default Value = 20	Real-time task. Four threads created, one for each channel. The threads wait for data to be read from the socket associated with the channel. Optionally, single thread for multiple sockets can be created instead of four threads to perform socket operations. Refer to the *Release Notes* for the procedures to enable single socket thread.

To enforce real-time behavior, it is important that `DspCtrlTsk` never takes too much time in any 10-ms period. Although the DSP solution is designed to avoid the burst execution in the control task, it can still be affected by the user applications.

For the performance and reliability reasons, it is suggested the user applications that are non-time-critical such as call-control and call-progress modules be implemented in Linux user-mode. The demo code in the DSP solution release provides an example of the client driver module.

As the middleware, the primary responsibility of the driver module is to act as a transport layer between the DSP solution's control interface and the user application and between the packet interface and the IP stack. The secondary responsibility is to perform the module initialization, which can be done as part of driver module initialization function. Additionally, the driver may also consolidate the messages and events from SLIC and other related modules into the same format and through a single queue to the user applications.

Figure 14. Intel® Infrastructure DSP Solution Client Driver in Linux*



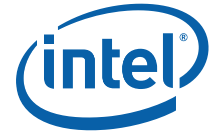
The driver can be implemented as an active or passive transport layer. In active mode, the driver spawns a dedicated kernel thread pending on the out-bound queue and automatically pumps the messages to applications once there is a message in the queue. In passive mode, it retrieves the message from the queue once the user application requests it.

The applications should not send a burst of control messages without waiting for the replies, or the real-time behavior of the DSP solution may be affected.

If the user application has to create kernel threads for time-critical data processing, the execution of the threads must be predictable and not impact the internal real time thread. As a guideline, the total execution time of these other threads should not exceed 1 ms in any 10-ms period. The DSP application in Linux* is depicted in Figure 14.

7.0 User-Defined Messages

To enable a simpler and more efficient interface, the DSP solution provides a facility for you to define custom messages, based on a combination of basic messages.



7.1 Overview

To enable the user control message facility using the Message Agent, the user application needs to register a decoder function and an encoder function with the DSP solution via the function `xDspSysInit()`. The decoder function is called by the DSP solution to decode all user control messages. The encoder function is called to handle all the replies to the decoder function, eventually encoding a reply message to the user message.

User control messages have the same format as the basic control messages, which contain a message header defined as:

```
typedef struct{
    UINT32   transactionId;/* used by apps to track the message */
    UINT16   instance;     /* instance ID (1-0xffff), 0:reserved */
    UINT8    resource;     /* MPR resource type */
    UINT8    reserved;     /* reserved for future */
    UINT16   size;         /* total size in bytes */
    UINT8    type;         /* message type */
    UINT8    attribute;    /* attribute, reserved for future */
} XMsgHdr_t, *XMsgRef_t;
```

The resource field in the header should specify `XMPR_MA`, which directs it to the Message Agent resource.

The instance field must always be 1, and because the field is always 1 for the messages sent to and received from the Message Agent, use the `transactionId` field to track the messages associated with the channels.

The type field in the header specifies type of message. User control messages should start with the value `XMSG_USRMSG_TYPE_BEGIN` - values less than this constant represents the basic control messages.

User-defined messages are delivered in the same way as the DSP solution control messages - using the same message queues for input and output, respectively.

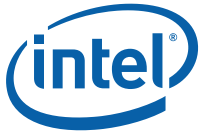
The Message Agent calls the registered decoder function when the type is beyond its internal range. The user-supplied decoder function is of the format:

```
typedef int (*XMsgAgentDec_t)(XMsgRef_t pUsrMsg,
                 XMsgRef_t pNativeMsg,  int sequenceNo);
```

The first parameter of the decoder function is the input message pointer, which references the control message to be decoded.

The second parameter is the output message pointer, which references the output of the decoder function. The `sequenceNo` field starts at 1 for the first decoder function call, and is incremented each time the decoder function is called.

The return value of the decoder function indicates whether the decoder function is complete with its message sequencing (returns 0); or whether the decoder function should continue to be called (returns non-zero). The return value can simply be the



number of messages left to sequence. If there is an error in the decoder process, the return value is set to negative. The return value - together with the sequence number - are used to drive the decoder function until the entire message sequence required is complete.

One useful feature of the Message Agent is the ability to recursively call other user control messages (maximum level of recursion is 4). This allows more complex functions to be built compactly.

The user encoder function is of the format:

```
typedef void (*XMsgAgentEnc_t)(XMsgRef_t pUsrReply, XMsgRef_t pNativeReply,

                          int sequenceNo, UINT8 usrMsgType);
```

The first parameter of the encoder function is the output message pointer, referencing the output of the encoder function.

The second parameter is the input message pointer, referencing the reply message from the resource component involved.

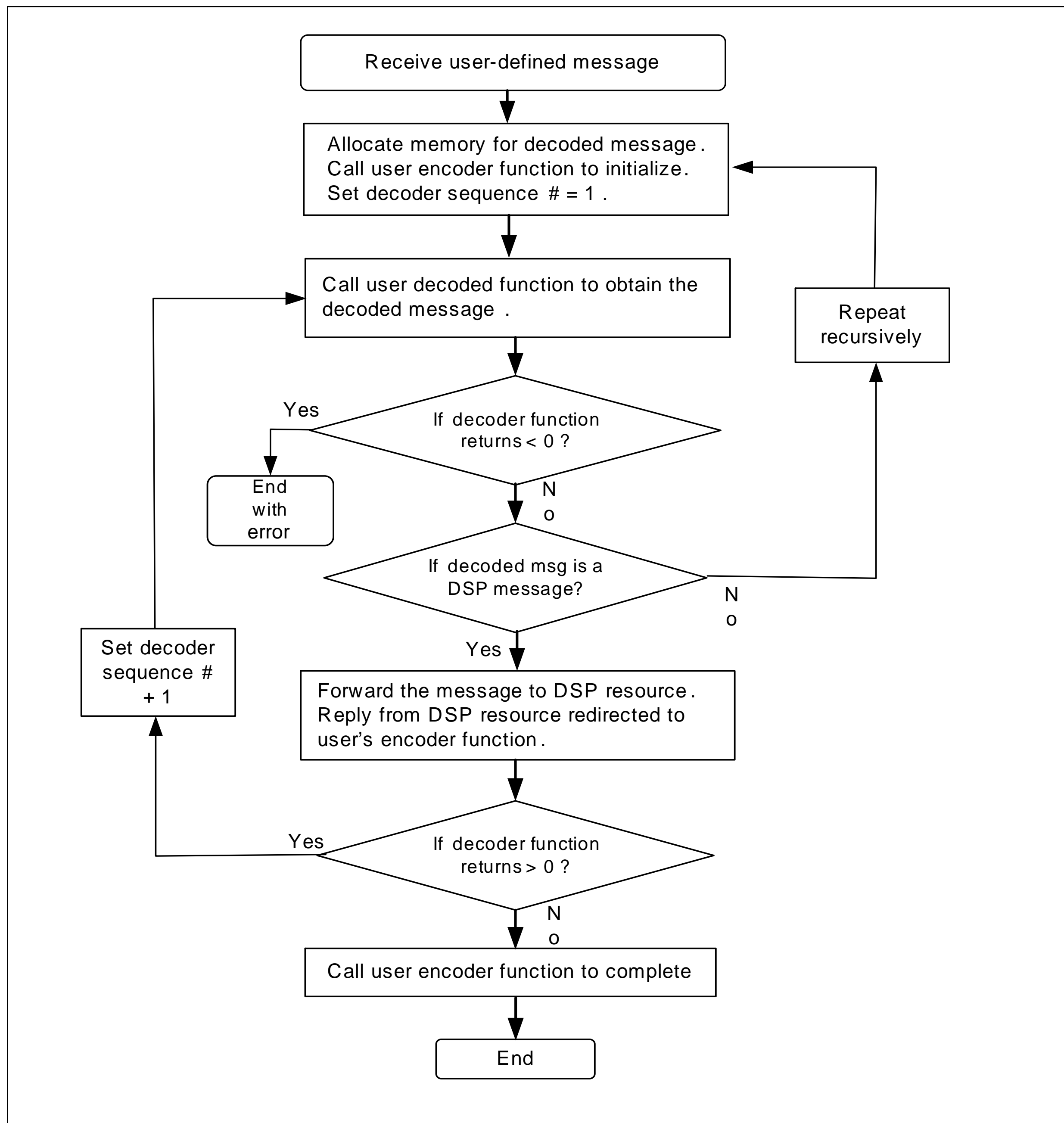
The Message Agent first calls the encoder function once with `sequenceNo` set to `XMSG_MA_ENCODING_INIT (0)` before receive the replies. Then the sequenceNo field is incremented each time a reply is received. The Message Agent in the DSP solution sets this field to `XMSG_MA_ENCODING_CMPLT (-1)` when the decoding process is complete.

The `usrMsgType` field informs the encoder of the user message ID, such that specific encoder functions may be called accordingly.

The replies to the decoded messages are re-directed to the encoder function, which can record the number of replies and any errors that may occur. A final reply message will be encoded and sent back to the user application when the message decoding process is complete.

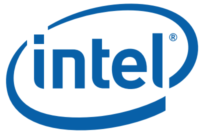
Figure 15 depicts how the Message Agent processes user-defined control messages.

Figure 15. Decoding User-Defined Messages in the Message Agent



7.2 Pre-Defined User Messages

This section describes the user messages that have already been implemented as examples. They can be further extended or modified by you. These messages form a higher-level control interface for the application scenarios like call setup, call transfer and three-way call. The control entities of this interface are the terminations which can be a TDM or IP terminations or a port of the mixer. The termination is specified by its type and channel defined as:



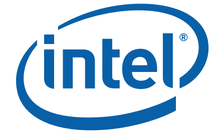

```
typedef struct{
  UINT8 type;
  UINT8 channel;
} __attribute__ ((packed)) IxDspCodeletTerm;
```

where channel is the channel number, and type can be:

```
typedef enum{
  IX_DSP_CODELET_TERM_NULL = 0,  /* null termination, to link to null
        means to disconnect the L-Port */
  IX_DSP_CODELET_TERM_TDM,  /* TDM termination contains one DSP
        dspResource - Network Endpoint which
        has a T-Port and a L-Port */
  IX_DSP_CODELET_TERM_IP,   /* IP termination contains DEC,ENC,
        TG and TD resources. It has one
        T-Port shared by DEC and TG and
        2 L-Ports for ENC and TD. But in
        This API, the 2 L-Ports always
        listen to the same talker */
  IX_DSP_CODELET_TERM_MIXER_PORT, /* Mixer termination has multiple
        T-Ports and L-Ports */
  IX_DSP_CODELET_TERM_EOL   /* End of List */
} IxDspCodeletTermType;
```

The following message types are defined and the corresponding message decoder and encoder functions are implemented:

```
typedef enum{
  /*------ messages send to Message Agent -------*/
  IX_DSP_CODELET_MSG_LINK = IX_DSP_CODELET_MSG_TYPE_BEGIN,
  IX_DSP_CODELET_MSG_LINK_BREAK,
  IX_DSP_CODELET_MSG_LINK_SWITCH,
  IX_DSP_CODELET_MSG_START_IP,
  IX_DSP_CODELET_MSG_STOP_IP,
  IX_DSP_CODELET_MSG_SETUP_CALL,
  IX_DSP_CODELET_MSG_SET_CALL_PARMS,
  IX_DSP_CODELET_MSG_SETUP_CALLWPARMS,
  IX_DSP_CODELET_MSG_SWITCH_CALL,
```



```
  IX_DSP_CODELET_MSG_CREATE_3WCALL,
  IX_DSP_CODELET_MSG_EXIT_3WCALL,
  IX_DSP_CODELET_MSG_TEARDOWN_3WCALL,
  IX_DSP_CODELET_MSG_BACKTO_2WCALL,
  IX_DSP_CODELET_MSG_SET_CLEAR_CHAN,
  IX_DSP_CODELET_MSG_T38_SWITCH,
  IX_DSP_CODELET_MSG_SET_PARMS,
  IX_DSP_CODELET_MSG_END_OF_OUTMSG,
  /*------messages received from Message Agent------*/
  IX_DSP_CODELET_MSG_ACK,
  IX_DSP_CODELET_MSG_LINK_ACK,
  IX_DSP_CODELET_MSG_SETUP_ACK,
  IX_DSP_CODELET_MSG_3W_ACK,
  IX_DSP_CODELET_MSG_t38_ACK,
  IX_DSP_CODELET_MSG_STOP_ACK,
  IX_DSP_CODELET_MSG_END_OF_LIST
} IxDspCodeletMsgType;
```

7.2.1 Link Message

Type: IX_DSP_CODELET_MSG_LINK

Direction: Inbound

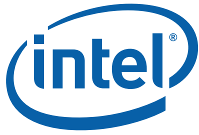
Description: Connects two specified terminations. Since terminations involve multiple resources, this involves multiple basic control messages.

Format:

```
typedef struct{
  XMsgHdr_t  header;
  IxDspCodeletTerm  term1;
  IxDspCodeletTerm  term2;
} IxDspCodeletMsgLink;
```

Macro:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_LINK(pMsg, trans) \
  {\
  XMSG_MA_MAKE_HEADER \
  (  pMsg, \
```




```
trans, \
IX_DSP_CODELET_MSG_LINK, \
sizeof(IxDspCodeletMsgLink) \
)\
}
```

Response: General Acknowledgement message (`IX_DSP_CODELET_MSG_LINK_ACK`)

7.2.2 Link Break Message

Type: `IX_DSP_CODELET_MSG_LINK_BREAK`

Direction: Inbound

Description: Disconnect two terminations. This connects each termination to null, using the `IX_DSP_CODELET_MSG_LINK` user message.

Format:

```
typedef struct{
    XMsgHdr_t            header;
    IxDspCodeletTerm     term1;
    IxDspCodeletTerm     term2;
} IxDspCodeletMsgLinkBreak;
```

Macro:

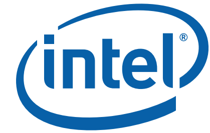
```
#define IX_DSP_CODELET_MAKE_MSGHDR_LINK_BREAK(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_LINK_BREAK, \
            sizeof(IxDspCodeletMsgLinkBreak) \
        )\
    }
```

Response: General Acknowledgement message (`IX_DSP_CODELET_MSG_LINK_ACK`)

7.2.3 Link Switch Message

Type: `IX_DSP_CODELET_MSG_LINK_SWITCH`

Direction: Inbound



Description: Disconnects the termination from one termination and connects to another. This connects the term termination to the switchTo termination, and connects the switchFrom termination to null. Again, this uses the `IX_DSP_CODELET_MSG_LINK` user message.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    term;
    IxDspCodeletTerm    switchFrom;
    IxDspCodeletTerm    switchTo;
} IxDspCodeletMsgLinkSwitch;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_LINK_SWITCH(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_LINK_SWITCH, \
            sizeof(IxDspCodeletMsgLinkSwitch) \
        )\
    }
```

Response: General Acknowledgement message (`IX_DSP_CODELET_MSG_LINK_ACK`)

7.2.4 Start IP Message

Type: `IX_DSP_CODELET_MSG_START_IP`

Direction: Inbound

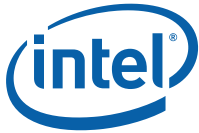
Description: Starts an IP termination. This involves the basic messages to start the Encoder, Decoder, and Tone Detector, respectively, and to stop the Tone Generator.

Format:

```
typedef struct{
    XMsgHdr_t   header;
    UINT8       channel;
} IxDspCodeletMsgStartIP;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSG_START_IP(pMsg, trans, chanIP) \
    {\
```




```
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_START_IP, \
            sizeof(IxDspCodeletMsgStartIP) \
        )\
        ((IxDspCodeletMsgStartIP *)(pMsg))->channel = (chanIP);\
    }
```

Response: General Acknowledgement message(IX_DSP_CODELET_MSG_SETUP_ACK)

7.2.5 Stop IP Message

Type: IX_DSP_CODELET_MSG_STOP_IP

Direction: Inbound

Description: Stops an IP termination. This involves the messages to stop the Encoder, Decoder, Tone Detector, and Tone Generator, respectively.

Format:

```
typedef struct{
    XMsgHdr_t   header;
    UINT8       channel;
} IxDspCodeletMsgStopIP;
```

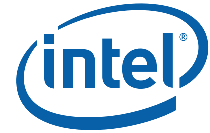
Macros:

```
#define IX_DSP_CODELET_MAKE_MSG_STOP_IP(pMsg, trans, chanIP) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_STOP_IP, \
            sizeof(IxDspCodeletMsgStopIP) \
        )\
        ((IxDspCodeletMsgStopIP *)(pMsg))->channel = (chanIP);\
    }
```

Response: Stop Acknowledgement message (IX_DSP_CODELET_MSG_STOP_ACK)

7.2.6 Set Up Call Message

Type: IX_DSP_CODELET_MSG_SETUP_CALL



Direction: Inbound

Description: Sets up a call. This uses two user messages, `IX_DSP_CODELET_MSG_LINK` to connect an HSS termination to an IP termination, and `IX_DSP_CODELET_MSG_START_IP` to start the IP termination.

Format:

```
typedef struct{
    XMsgHdr_t   header;
    UINT8       channelIP;
    UINT8       channelTDM;
} IxDspCodeletMsgSetupCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSG_SETUP_CALL(pMsg, trans, chanIP, chanTDM) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SETUP_CALL, \
            sizeof(IxDspCodeletMsgSetupCall) \
        )\
        ((IxDspCodeletMsgSetupCall *)(pMsg))->channelIP = (chanIP);\
        ((IxDspCodeletMsgSetupCall *)(pMsg))->channelTDM = (chanTDM);\
    }
```

Response: General acknowledgement message(`IX_DSP_CODELET_MSG_SETUP_ACK`)

7.2.7 Set Call Parameters Message

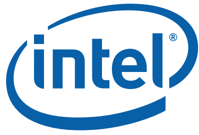
Type: `IX_DSP_CODELET_MSG_SET_CALL_PARMS`

Direction: Inbound

Description: Sets parameters of a call. These parameters are likely affected by the results of negotiation between the call stacks and may change call by call. The message involves four basic messages to set the parameters for the Encoder, Decoder, Tone Detector, and Tone Generator of an IP termination.

Format:

```
typedef struct{
    XMsgHdr_t               header;
    IxDspCodeletCallParms   parms;
    UINT8                   channelIP;
```




```
}IxDspCodeletSetCallParms;
```

where `IxDspCodeletCallParms` is defined as:

```
typedef struct{
    UINT16  decAutoSwitch;
    UINT8   decType;
    UINT8   encType;
    UINT8   frmsPerPkt;
    UINT8   vad;
    UINT8   rfc2833;
    UINT8   rfc2833pyldType;
    UINT8   toneClamp;
} IxDspCodeletCallParms;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_SET_CALL_PARMS(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SET_CALL_PARMS, \
            sizeof(IxDspCodeletSetCallParms) \
        )\
    }
```

Response: General acknowledgement message (`IX_DSP_CODELET_MSG_ACK`)

7.2.8 Set Up Call with Parameters Message

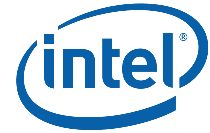
Type: `IX_DSP_CODELET_MSG_SETUP_CALLWPARMS`

Direction: Inbound

Description: Set up a call with parameters. This involves two user messages, `IX_DSP_CODELET_MSG_SET_CALL_PARMS` to set up the call parameters, and `IX_DSP_CODELET_MSG_SETUP_CALL` to set up the call.

Format:

```
typedef struct{
    XMsgHdr_t               header;
    IxDspCodeletCallParms   parms;
    UINT8                   channelIP;
```



```
    UINT8                    channelTDM;
} IxDspCodeletMsgSetupCallwParms;
```

where `IxDspCodeletCallParms` is defined as:

```
typedef struct{
    UINT16  decAutoSwitch;
    UINT8   decType;
    UINT8   encType;
    UINT8   frmsPerPkt;
    UINT8   vad;
    UINT8   rfc2833;
    UINT8   rfc2833pyldType;
    UINT8   toneClamp;
} IxDspCodeletCallParms;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_SETUP_CALLWPARMS(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SETUP_CALLWPARMS, \
            sizeof(IxDspCodeletMsgSetupCallwParms) \
        )\
    }
```

Response: General acknowledgement message(`IX_DSP_CODELET_MSG_SETUP_ACK`)

7.2.9 Switch Call Message

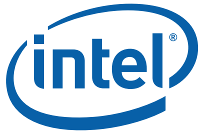
Type: `IX_DSP_CODELET_MSG_SWITCH_CALL`

Direction: Inbound

Description: Switches a call. This involves two user messages, `IX_DSP_CODELET_MSG_LINK_SWITCH` to switch an HSS termination to another IP termination, and `IX_DSP_CODELET_MSG_SETUP_CALL` to set up the call.

Format:

```
typedef struct{
    XMsgHdr_t   header;
    UINT8       channelTDM;
```




```
    UINT8        ipChanOnHold;
    UINT8        ipChanNewCall;
} IxDspCodeletMsgSwitchCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSG_SWITCH_CALL(pMsg, trans, chTDM, chHld, chNew) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SWITCH_CALL, \
            sizeof(IxDspCodeletMsgSwitchCall) \
        )\
        ((IxDspCodeletMsgSwitchCall *)(pMsg))->channelTDM = (chTDM);\
        ((IxDspCodeletMsgSwitchCall *)(pMsg))->ipChanOnHold = (chHld);\
        ((IxDspCodeletMsgSwitchCall *)(pMsg))->ipChanNewCall = (chNew);\
    }
```

Response: General acknowledgement message (`IX_DSP_CODELET_MSG_LINK_ACK`)

7.2.10 Create Three-Way Call Message

Type: `IX_DSP_CODELET_MSG_CREATE_3WCALL`

Direction: Inbound

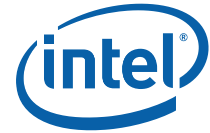
Description: Sets up a three-way call. This involves using user message `IX_DSP_CODELET_MSG_LINK` three times to connect each of the three parties in the three-way call to the mixer. Then a basic message is used to start the mixer resource.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    parties[3];
} IxDspCodeletMsgCreate3wCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_CREATE_3WCALL(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
```



```
                IX_DSP_CODELET_MSG_CREATE_3WCALL, \
                sizeof(IxDspCodeletMsgCreate3wCall) \
            )\
    }
```

Response: General acknowledgement message (`IX_DSP_CODELET_MSG_3W_ACK`).

7.2.11 Exit Three-Way Call Message

Type: `IX_DSP_CODELET_MSG_EXIT_3WCALL`

Direction: Inbound

Description: Exits a three-way call. This is the same as in `IX_DSP_CODELET_MSG_CREATE_3WCALL`, except the `IX_DSP_CODELET_MSG_LINK_BREAK` is used instead. Then a basic message is used to stop the mixer resource.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    parties[3];
} IxDspCodeletMsgExit3wCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_EXIT_3WCALL(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_EXIT_3WCALL, \
            sizeof(IxDspCodeletMsgExit3wCall) \
        )\
    }
```

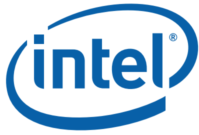
Response: General acknowledgement message (`IX_DSP_CODELET_MSG_3W_ACK`)

7.2.12 Teardown Three-Way Call Message

Type: `IX_DSP_CODELET_MSG_TEARDOWN_3WCALL`

Direction: Inbound

Description: Teardown a three-way call. This involves first using the user message `IX_DSP_CODELET_MSG_EXIT_3WCALL` to exit the three-way call. Then the user message `IX_DSP_CODELET_MSG_STOP_IP` is used to stop any IP channels that have been connected.



Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    parties[3];
} IxDspCodeletMsgTeardown3wCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_TEARDOWN_3WCALL(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_TEARDOWN_3WCALL, \
            sizeof(IxDspCodeletMsgTeardown3wCall) \
        )\
    }
```

Response: Stop acknowledgement message (`IX_DSP_CODELET_MSG_STOP_ACK`)

7.2.13 Back to Two-Way Call Message

Type: `IX_DSP_CODELET_MSG_BACKTO_2WCALL`

Direction: Inbound

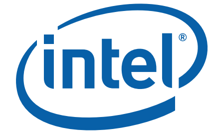
Description: Changes a three-way call to a two-way call. It involves using the user message `IX_DSP_CODELET_MSG_EXIT_3WCALL` to exit the three-way call. Then the user message `IX_DSP_CODELET_MSG_LINK` is used to create the two-way call. Then the user message `IX_DSP_CODELET_MSG_STOP_IP` is used to stop the IP termination if the disconnected party is one.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    party1;
    IxDspCodeletTerm    party2;
    IxDspCodeletTerm    partyToDrop;
} IxDspCodeletMsgBackto2wCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_BACKTO_2WCALL(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
```



```
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_BACKTO_2WCALL, \
            sizeof(IxDspCodeletMsgBackto2wCall) \
        )\
    }
```

Response: General acknowledgement message (`IX_DSP_CODELET_MSG_3W_ACK`).

7.2.14 Set Clear Channel Message

Type: `IX_DSP_CODELET_MSG_SET_CLEAR_CHAN`

Direction: Inbound

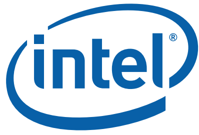
Description: Sets a channel to clear channel. This involves five basic messages to set the parameters of the Encoder, Decoder, Tone Generator, Tone Detector, and Network resources, respectively.

Format:

```
typedef struct{
    XMsgHdr_t   header;
    UINT8       channelIP;
    UINT8       channelTDM;
    UINT8       codeType;
} IxDspCodeletMsgSetClearChan;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSG_SET_CLEAR_CHAN(pMsg, trans, chanIP, ChanTDM, code)
\
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SET_CLEAR_CHAN, \
            sizeof(IxDspCodeletMsgSetClearChan) \
        )\
        ((IxDspCodeletMsgSetClearChan *)(pMsg))->channelIP = chanIP; \
        ((IxDspCodeletMsgSetClearChan *)(pMsg))->channelTDM = ChanTDM; \
        ((IxDspCodeletMsgSetClearChan *)(pMsg))->codeType = code; \
    }
```



Response: General acknowledgement message (`IX_DSP_CODELET_MSG_ACK`)

7.2.15 T.38 Switchover Message

- **Type** : `IX_DSP_CODELET_MSG_T38_SWITCH`
- **Direction** : Inbound
- **Description** : Switches a channel between voice and T.38 fax modes
- **Format** :

```
typedef struct{
 XMsgHdr_t      header;
 UINT8          channelIP;
 UINT8          channelTDM;
 UINT8          mode;           /* mode to switch, fax or voice */
 UINT8          tone;           /* Tone id that triggered switch */
 }IxDspCodeletMsgT38Switch;
```

- **Macros** :

```
#define IX_DSP_CODELET_MAKE_MSG_T38_SWITCH(pMsg, trans,
chanIP,ChanTDM,md, toneId) \
{\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_T38_SWITCH, \
            sizeof(IxDspCodeletMsgT38Switch) \
        )\
        ((IxDspCodeletMsgT38Switch *)(pMsg))->channelIP = chanIP; \
        ((IxDspCodeletMsgT38Switch *)(pMsg))->channelTDM = ChanTDM; \
        ((IxDspCodeletMsgT38Switch *)pMsg)->mode = md;\
        ((IxDspCodeletMsgT38Switch *)pMsg)->tone = toneid;\
}
```

- **Response:**
  T.38 acknowledgement message (`IX_DSP_CODELET_MSG_T38_ACK`)

7.2.16 Set Parameters Message

Type: `IX_DSP_CODELET_MSG_SET_PARMS`

Direction: Inbound

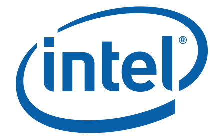
Description: Sets parameters. It sends basic messages to set the parameters from an input list across the different resource components involved.

Format:

```
typedef struct{

    XMsgHdr_t      header;

    UINT16         numParms;

    IxDspCodeletParm parms[IX_DSP_CODELET_MAX_PARMS];

} IxDspCodeletMsgSetParms;
```

where `IxDspCodeletParm` is defined as:

```
typedef struct{

    UINT16  parmID;
```



```
    INT16    value;
    UINT8    dspResource;
    UINT8    dspResInstance;
} __attribute__ ((packed)) IxDspCodeletParm;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_SET_PARMS(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SET_PARMS, \
            sizeof(IxDspCodeletMsgSetParms) \
        )\
    }
```

Response: General acknowledgement message (`IX_DSP_CODELET_MSG_ACK`).

7.3 Pre-Defined User-Response Messages

7.3.1 Acknowledge Message

There are three Acknowledge messages that are of the same format, but correspond to different control messages.

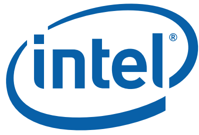
Type: `IX_DSP_CODELET_MSG_ACK, IX_DSP_CODELET_MSG_LINK_ACK, IX_DSP_CODELET_MSG_SETUP_ACK` and `IX_DSP_CODELET_MSG_T38_ACK`.

Direction: Outbound

Description: Acknowledge messages to user control messages.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    INT16               numDspReplies;
    INT16               numErrors;
    IxDspCodeletError   error[IX_DSP_CODELET_MAX_ERR_REPLY];
} IxDspCodeletMsgAck,
IxDspCodeletMsgLinkAck,
IxDspCodeletMsgSetupAck,
IxDspCodeletMsgT38Ack;
```



where `IxDspCodeletError` is defined as:

```
typedef struct{
    UINT32  errData;
    UINT16  errCode;
    UINT8   dspResource;
    UINT8   dspResInstance;
} IxDspCodeletError;
```

7.3.2 Stop Acknowledge Message

Type: `IX_DSP_CODELET_MSG_STOP_ACK`

Direction: Outbound

Description: Stops acknowledge message to user stop messages.

Format:

```
typedef struct{
    XMsgHdr_t               header;
    INT16                   numDspReplies;
    INT16                   numErrors;
    IxDspCodeletError       error[IX_DSP_CODELET_MAX_ERR_REPLY];
    INT16                   numStopAck;
    IxDspCodeletStopCmplt stopAck[IX_DSP_CODELET_MAX_STOP_CMPLT];
} IxDspCodeletMsgStopAck;
```

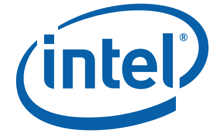
where `IxDspCodeletError` is defined above and `IxDspCodeletStopCmplt` is defined as:

```
typedef struct{
    UINT32  totalFrames;
    UINT8   dspResource;
    UINT8   dspResInstance;
} IxDspCodeletStopCmplt;
```

8.0 Application Examples

8.1 IP Interface

The DSP solution uses two interface functions to transfer encoded audio packets to and from the IP interface. These audio packets are transferred on the IP network as RTP (Real-time Transport Protocol) packets. RTP packets are UDP (User Datagram Protocol)



packets with a 12-byte RTP header at the beginning of the UDP payload. UDP packets are suitable for transmitting real-time media data since they are low on overhead and thus provide speedy delivery, though packet delivery is not guaranteed.

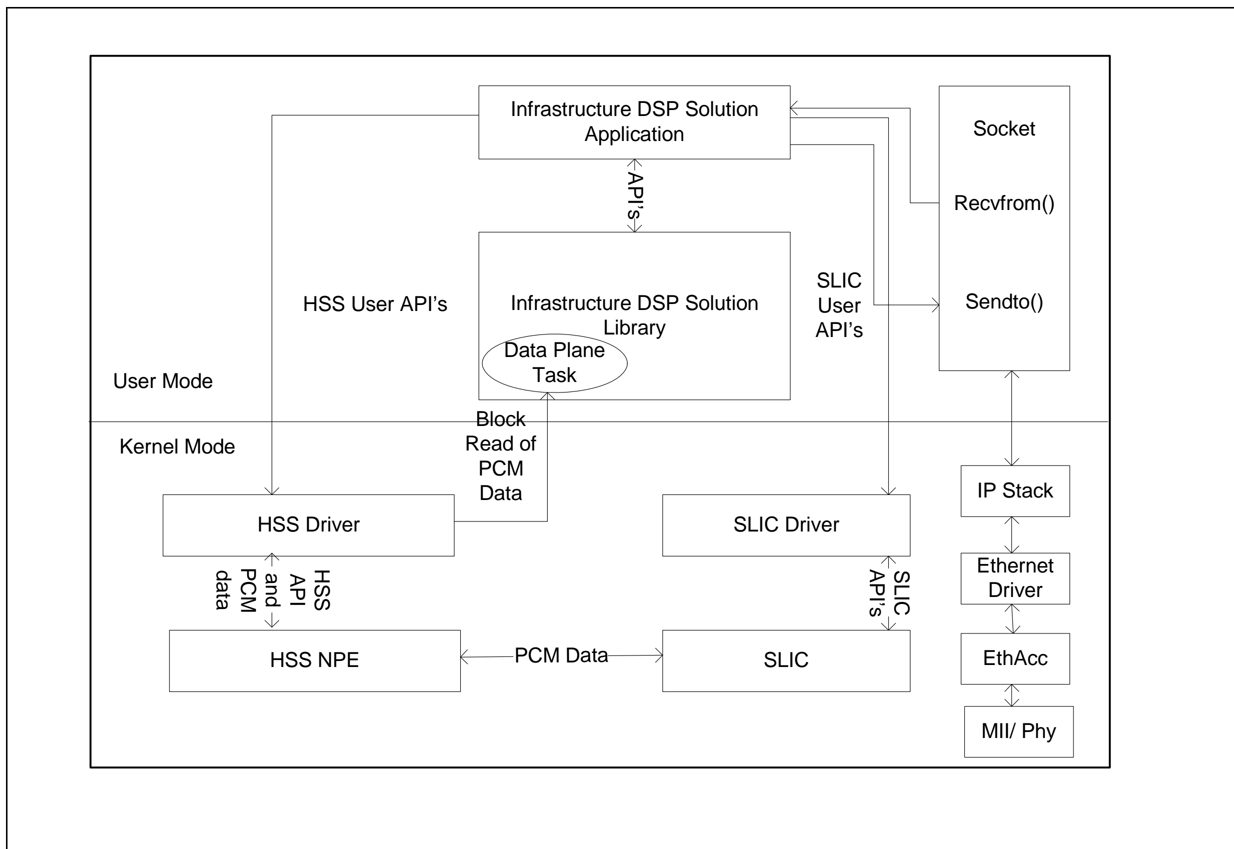
The RTP packet streams must be extracted from the overall incoming IP traffic in the ingress direction, and merged to outgoing IP traffic in the egress direction. One way to do this is to examine the IP packets in the Ethernet driver. Incoming RTP packets are routed to the DSP solution, while other IP packets are sent to the user's IP stack. Another way is to route all IP traffic to the user application from the Ethernet driver. Then use standard interfaces, such as sockets, to route the appropriate traffic to the respective parties.

The advantage of the second approach is that socket functions are already provided by the Linux operating system. The EthAcc interface of the DSP solution is integrated with the Ethernet driver and the user application developers need not worry about this service. The user application is only required to perform initialization and then exchange control messages and data packets between the DSP solution and the application. Because an application can open multiple sockets as needed, the resource in the DSP solution can be shared by multiple clients.

A typical VoIP application using sockets will consist of a few tasks that handle either the data packets or the control messages. Figure 16 shows the possible implementation in Linux.

The application may choose to send and receive packets through sockets with a single port. A channel ID can be embedded in the package so that the package can be passed to the corresponding the DSP solution channel based on the ID number. Alternatively, the application may choose to map the socket port number with the DSP solution channel.

Figure 16. Intel® Infrastructure DSP Solution Application in Linux*



8.2 Caller-ID Generator

The FSK modulator provided in the DSP solution is designed primarily to allow user applications to implement the caller-ID generator.

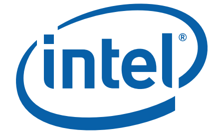
Note: The caller-ID generation is a function of the user application as it involves direct interaction with the specific SLIC interface being used.

The caller ID specifications are country-specific and some of them can be found in the documents of Bellcore* 202 for the United States, Technical Specification YDN 069-1977 for China, and the NTT Technical Reference - Telephone Interfaces, Edition 5 for Japan.

In this release, the demo source codes are included to show how to implement U.S., China, and Japan caller-ID generators on the evaluation platform using the FSK feature according to these specifications.

To implement caller-ID generation, the user's applications are responsible to provide the following functions in addition to FSK modulator:

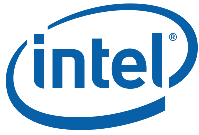
- Generate the complete caller ID data to be transmitted by the FSK modem. The data must be represented in octet (byte) without mark, start, and stop bits. The demo code includes useful utilities that can build the caller ID data format from the information to be displayed and add parity check bits, CRC octets, or check sum if necessary.



- Control the SLIC device to generate the signals such as polarity reverse, short ring (CRA), and normal ring as required by the caller-ID specifications
- Detect the loop connection/disconnection (or off-hook/on-hook status) for Japan caller ID. SLIC driver may report such events through the outbound message queue using the complementary function of hook-event detection.
- Provide timer service using OS services, based on hardware or software resources. The built-in complementary timer service function in the NET component in the DSP solution can be used for this purpose. The timer events can be reported through the outbound message queue.
- Implement the state machine that follows the signal flow diagram of the caller ID as described in NTT* specifications. The data ID data as we discussed above are transmitted using the FSK modem function in the proper state. The demo code gives an example of such state machine.

Most of other country-specific caller-ID generators can be implemented similarly. Some caller-ID specifications, like Japan, require the FSK data to be transmitted in off-hook state, while others transmit the data in on-hook state. The procedure of on-hook transmission is simpler because the interactions between SLIC device and the caller-ID receiver are no longer necessary.

§ §

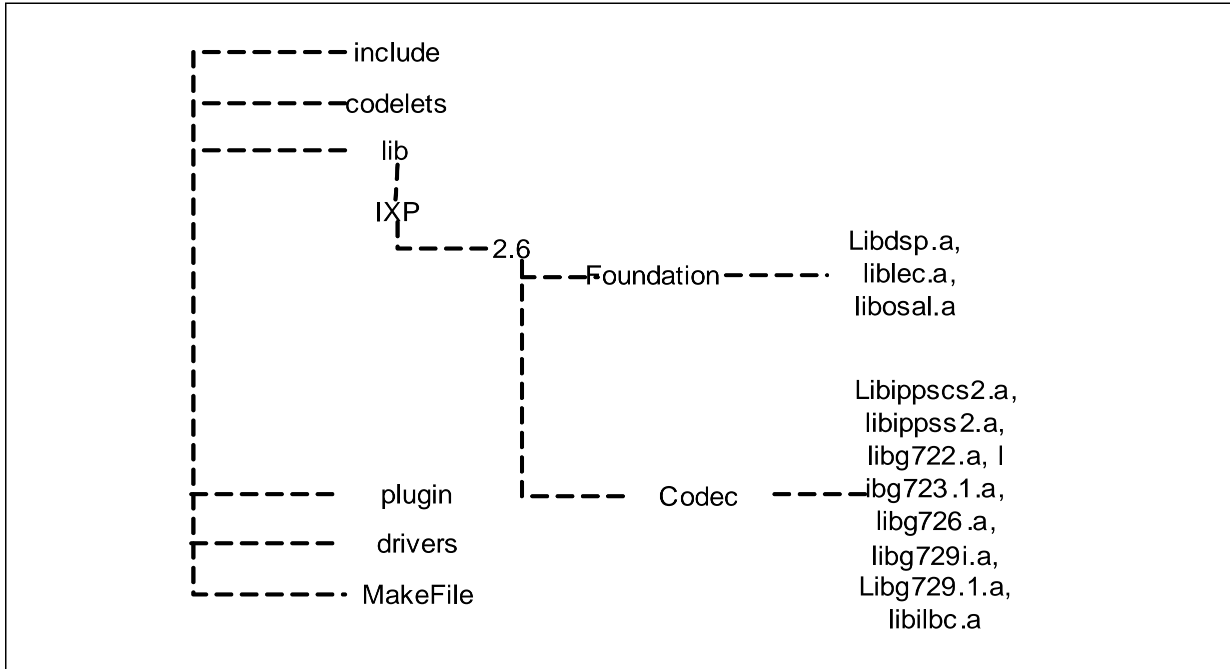


Appendix A : Procedure to Add Intel-Provided DSP Plug-ins

This section gives the procedure to add and build Intel-provided plug-ins.

A.1 Directory Structure

The Intel® Infrastructure DSP Solution package consists of the following directory structures.



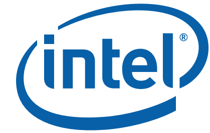
The plug-in folder contains the plug-in configuration files.

Component	Description
`PlugInConfig.h`	This is mainly used to define all plug-ins header files.
`PlugInConfig.c`	Configures the CODECs/EC plug-ins. Adds entry for either CODEC or EC (in any order) plug-ins in the plug-in list.

A.2 Edit PlugInConfig.c

Edit `IDS/plugin/PlugInConfig.c` as below:

```
PlugInConfig.c

#include "PlugInConfig.h" /*The global PlugInList*/

PLUG_IN_LIST_BEGIN

    /********Don't edit or modify above this line********/

    /*Add plug-in entries below */
```



```
    USCI_CODEC_PLUG_IN (USC_G722SB_Fxns,XCODER_TYPE_G722,0)
    USCI_CODEC_PLUG_IN(USC_G729A_Fxns,XCODER_TYPE_G729A,0)
    USCI_CODEC_PLUG_IN(USC_G726_Fxns,XCODER_TYPE_G726_40,0)
    USCI_CODEC_PLUG_IN(USC_G726_Fxns,XCODER_TYPE_G726_32,0)
    USCI_CODEC_PLUG_IN(USC_G726_Fxns,XCODER_TYPE_G726_24,0)
    USCI_CODEC_PLUG_IN(USC_G726_Fxns,XCODER_TYPE_G726_16,0)
    USCI_CODEC_PLUG_IN(USC_G723_Fxns,XCODER_TYPE_G723,0)
    USCI_CODEC_PLUG_IN(USC_G7291_Fxns,XCODER_TYPE_G729_1,0)
    USCI_CODEC_PLUG_IN(USC_ILBC_Fxns,XCODER_TYPE_ILBC_30MS,0)
    USCI_CODEC_PLUG_IN(USC_ILBC_Fxns,XCODER_TYPE_ILBC_20MS,0)
    USCI_FAX_PLUG_IN(USC_T38INT_Fxns,XFAX_TYPE_T38,0)
    /*Echo cancellor plug_ins*/
    USCI_EC_PLUG_IN(USC_LECINT_Fxns,XEC_TYPE_LINE_EC,0)
    /********Don't edit or modify beyond this line********/
PLUG_IN_LIST_END
```

A.3 Examples

A.3.1 Case 1

To plug in only G726, rate-40Kbps, select:

`USCI_CODEC_PLUG_IN (USC_G726_Fxns,XCODER_TYPE_G726_40,0)` and comment all other lines between the lines `PLUG_IN_LIST_BEGIN` and `PLUG_IN_LIST_END`.

A.3.2 Case 2

To select line echo canceller, select:

`USCI_EC_PLUG_IN (USC_LECINT_Fxns,XEC_TYPE_LINE_EC,0)` and comment all other lines between the lines `PLUG_IN_LIST_BEGIN` and `PLUG_IN_LIST_END`.

A.3.3 Case 3

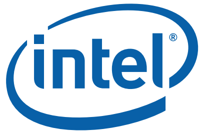
To select "G726, rate-40Kbps" and "line echo canceller", select the following two lines:

```
USCI_CODEC_PLUG_IN(USC_G726_Fxns,XCODER_TYPE_G726_40,0)

USCI_EC_PLUG_IN (USC_LECINT_Fxns,XEC_TYPE_LINE_EC,0)
```

A.4 Edit PlugInConfig.h

Edit the `IDS/plugin/PlugInConfig.h` as below:

```
PlugInConfig.h
```




```
#ifndef __PLUGINCONFIG_H__

#define __PLUGINCONFIG_H__

#include "PlugInDefs.h"

DECLARE_PLUG_IN_LIST

/*Don't edit or modify above this line*/

/*Add Plug-in Header file below*/

#include "g729usc.h"

#include "g723usc.h"

#include "g726usc.h"

#include "g722usc.h"

#include “g729_1usc.h”

#include “iLBCusc.h”

#include “t38interface.h”

#include "lecinterface.h" /*Line echo cancellor*/

/*Don't edit or modify beyond this line*/

#endif /*__PLUGINCONFIG_H__*/
```

A.5 Examples

A.5.1 Case 1

To plug-in only G726, rate-40Kbps, select:

`#include "g726usc.h"` and comment all other lines between the lines `DECLARE_PLUG_IN_LIST` and `#endif`

A.5.2 Case 2

To select line echo canceller, select:

`#include "lecinterface.h"` and comment all other lines between the lines `DCLARE_PLUG_IN_LIST` and `#endif`

A.5.3 Case 3

To select G726, rate-40Kbps and line echo canceller, select the following two lines

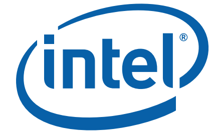
`#include "g726usc.h"`

`#include "lecinterface.h"`

and comment all other lines between the lines `DCLARE_PLUG_IN_LIST` and `#endif`

A.6 Building Plug-in Module

Build DSP plug-in module by using the command `make plug`. This builds the plug-in.



```
cd <workdir>/IDS

make plug
```

Appendix B : Adding Third Party Plug-ins

This section gives the procedure to plug-in a third party/external plug-in into the DSP solution. It also gives the procedure to unplug the plugged-in component.

Note: The third party plug-in codecs or external plug-in codecs must be the same codec type as the supported Intel-provided codec type.

B.1 Procedure to Add and Build Third Party Plug-ins

Copy the plug-in, for example, `libxyz.a` to the location `<Installed dir>/IDS/lib/IXP/<kernel version>/Codec`

Modify the plug-in configuration file `PlugInConfig.c` to hook the customer plug-in into DSP solution as explained in Appendix A.

Modify the Make file to link the plug-ins, For example, for plugging into the demo codelet application, modify the file `<Installed dir>/IDS/codelets/dspApp/Makefile` as shown in Figure 17. In case of plugging into customer applications, link the plug-in file libxyz.a in the application's make file in a similar way.

Figure 17. Snap-shot of Demo Codelet Make File

```
47
48 # Add third party or customer plug-ins here
49 ifeq ($(IX_LINUXVER),2.6)
50 TP_PLUGINS =     # -l<Plug_in1> -l<Plug_in2> -l<Plug_inN>
51 else
52 TP_PLUGINS =    -lxyz # -l<Plug_in1> -l<Plug_in2> -l -l<Plug_inN>
53 endif
54
```

Build `PlugInConfig.c` to generate file `libPlug*.a` using `<Installed dir>/IDS/plugin/Makefile` as explained in Appendix A.4.

Build demo codelet/customer application.

B.2 Procedure to Access Plug-in Parameters

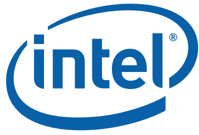
Two APIs are available to access plug-in parameters:

```
XStatus_t xDspPlugInParmRead(UINT8 res, UINT16 inst,UINT16
algType, UINT16 parmId, UINT32 *pParmVal);
XStatus_t xDspPlugInParmWrite(UINT8 res, UINT16 inst,UINT16
algType, UINT16 parmId, UINT32 ParmVal);
```

This section describes how to use parameter IDs (paramID) for these APIs.

For example, Echo Canceller (EC) is plugged in as a third party component that supports some additional custom parameters, say `paramX` and `paramY`.

The following are USCI defined parameters:




```
/* USC echo canceller modes */

typedef struct {

   USC_AdaptType adapt;      /* 0 - disable adaptation, 1 - enable full
adaptation, 2 - enable lite adaptation  */

   int zeroCoeff;  /* 0 - no zero coeffs of filters, 1 - zero coeffs of
filters  */

   int cng;        /* 0 - disable CNG, 1 - enable CNG */

   int nlp;        /* 0 - disable NLP, 1 or 2 - enable different NLP types
*/

   int td;         /* 0 - disable ToneDisabler, 1 - enable ToneDisabler */

   int ah;         /* 0 - disable anti-howling,  1-spectra-based HD, 2-
energy-based HD */

}USC_EC_Modes;
```

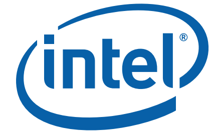
Here the USCI parameter index is a zero-based index from the structure member `adapt` till the member `freq_shift` in the USC_EC_Modes, as shown above.

This implies USCI parameter ID for all members, as shown below:

USCI Parameter ID	USC_EC_Modes Member
0	adapt
1	zeroCoeff
	cng
2	nlp
5	td
6	ah

To support additional custom parameters, say `ParamX` and `ParamY`, the implementation of USC_EC_Modes structure is as follows:

```
/* USC echo canceller modes */

typedef struct {

/* 0 - disable adaptation, 1 - enable full adaptation, 2 - enable lite adaptation
*/

   USC_AdaptType adapt;

   int      zeroCoeff;  /* 0 - no zero coeffs of filters,

                          1 - zero coeffs of filters  */

   int      nlp;       /* 0 - disable NLP, 1 - enable NLP */

   int      cng;        /* 0 - disable CNG, 1 - enable CNG */

   int      td;         /* 0 - disable ToneDisabler,

                          1 - enable ToneDisabler */

   int      ah;          /* 0 - disable anti-howling,
```



```
                                   1-spectra-based HD, 2- energy-based HD */
      int       paramX;
      int       paramY;
   }USC_EC_Modes;
```

Note: It is mandatory to add custom parameters at the end, as shown above.

The USCI ID for the custom parameter is as shown below.

USCI Parameter ID	USC_EC_Modes Member
0	adapt
1	zeroCoeff
	cng
2	nlp
5	td
6	ah
9	paramX
10	paramY

Applications can read/write the custom parameters paramX and paramY with the Parameter ID as 9 and 10.

B.3 Procedure to Un-Plug Third Party Plug-ins

Remove or comment out the plug-in entry in file PlugInConfig.c. An example is illustrated in Figure 18.

Figure 18. Snap-shot of PlugInConfig.c file Showing Changes Needed for Unplugging a Third Party Plug-in

```
 8 /*       The global PlugInList*/
 9 PLUG_IN_LIST_BEGIN
10
11         /********Don't edit or modify above this line********/
12         /*Add plug-in entries below */
13
14         USCI_CODEC_PLUG_IN(USC_G722SB_Fxns,XCODER_TYPE_G722,0)
15         USCI_CODEC_PLUG_IN(USC_G729A_Fxns,XCODER_TYPE_G729A,0)
16         USCI_CODEC_PLUG_IN(USC_G726_Fxns,XCODER_TYPE_G726_40,0)
17         USCI_CODEC_PLUG_IN(USC_G726_Fxns,XCODER_TYPE_G726_32,0)
18         USCI_CODEC_PLUG_IN(USC_G726_Fxns,XCODER_TYPE_G726_24,0)
19         USCI_CODEC_PLUG_IN(USC_G726_Fxns,XCODER_TYPE_G726_16,0)
20         USCI_CODEC_PLUG_IN(USC_G723_Fxns,XCODER_TYPE_G723,0)
21         USCI_CODEC_PLUG_IN(USC_G7291_Fxns,XCODER_TYPE_G729_1,0)
22         USCI_CODEC_PLUG_IN(USC_ILBC_Fxns,XCODER_TYPE_ILBC_20MS,0)
23         USCI_CODEC_PLUG_IN(USC_ILBC_Fxns,XCODER_TYPE_ILBC_30MS,0)
24         USCI_CODEC_PLUG_IN(USC_T38INT_Fxns,XFAX_TYPE_T38,0)
25         /*Echo cancellor plug_ins*/
26         USCI_EC_PLUG_IN(USC_LECINT_Fxns,XEC_TYPE_LINE_EC,0)
27
```

Remove the plug-in entry in the make file that is used for building the application. Figure 19 shows an example to remove entry in codelet demo make file.

Note: Strikethrough font in the figure illustrates that the text should be deleted.

Figure 19. Example to Remove a Plug-in in Codelet Demo Make File

```
47
48 # Add third party or customer plug-ins here
49 ifeq ($(IX_LINUXVER),2.6)
50 TP_PLUGINS =     # -l<Plug_in1> -l<Plug_in2> -l<Plug_inN>
51 else
52 TP_PLUGINS =    ~~lxyz~~ # -l<Plug_in1> -l<Plug_in2> -l -l<Plug_inN>
53 endif
54
```

Rebuild PlugInConfig.c to generate file libPlug*.a using <Installed dir>/IDS/plugin/Makefile as explained in Appendix A.4.

Rebuild and run the application or codelet demo application to reflect the unplug changes.

Note:

- — At present the DSP solution supports only CODECs and echo canceller plug-ins.
- — To be pluggable, the CODECs and echo canceller should be USCI compliant.
- — DSP solution supports only 10ms, 30ms linear PCM samples for CODECs and 8ms and 10ms frame sizes.
- — Plug-ins can only be statically plugged in. They cannot be plugged in at runtime.

Appendix C : Socket Interface

The user application uses an IP address-based call setup procedure by using the socket configuration menu. You can specify the destination IP address together with the calling channel and called channel information. The ixDspCodeletSocketChanConfig routine maps the source and destination channel numbers to IP address/UDP port combination, thereby providing the addressing information for the socket interface.

ixDspCodeletSocketCreate routine creates a pair of sockets per channel for transmit and receive operation. In the Ingress path, packets are received on the Rx sockets associated with the channel and are passed on to the DSP via the call to xPacketReceive function. On the Egress path, the ixDspCodeletRtpSocketSend is the callback routine that is registered by the application and called by the DSP solution to deliver packets to the network stack. This routine uses the addressing information to transmit packets on the network via the IP stack.

ixDspCodelet_RTP_Sendto() and ixDspCodelet_RTP_Recvfrom() are used in the egress and ingress paths respectively for RTP processing and SRTP encrypt/decrypt functions from the srtp library. Figure 20 illustrates this.

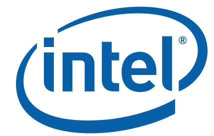


Figure 20. Socket Interface

Infrastructure DSP Solution
Infrastructure DSP Solution Codelet
Tx Socket
Rx Socket
Linux RTOS
IP
UDP Transport
Stack
User Space
Kernel Space
IXP400 Linux Ethernet Device Driver
ethAcc Component

§ §